NEXT GENERATION REAL-TIME SYSTEMS: INVESTIGATING THE POTENTIAL OF PARTIAL-SOLUTION TASKS

THESIS

Robert Edmund James Caley
Captain, USAF

AFIT/GCS/ENG/94D-01

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/94D-01

NEXT GENERATION REAL-TIME SYSTEMS: INVESTIGATING THE POTENTIAL OF PARTIAL-SOLUTION TASKS

THESIS

Robert Edmund James Caley
Captain, USAF

AFIT/GCS/ENG/94D-01

AFIT/GCS/ENG/94D-01

NEXT GENERATION REAL-TIME SYSTEMS: INVESTIGATING THE POTENTIAL OF PARTIAL-SOLUTION TASKS

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfilment of the

Requirements for the Degree of

Master of Science (Computer Science)

Robert Edmund James Caley, B.S.

Captain, USAF

December 1994

Acknowledgements

This thesis would not have been possible without the help of several individuals:

First and foremost I would like to thank my wife Kendle and my two sons Kyle and Brett for putting up with my preoccupation and repeated absences during the last eighteen months. Their patience and understanding allowed me to focus on research issues, while their subtle hints kept me in touch with the reality of family life.

Second, I would like to thank my research advisor Major Gregg Gunsch, who took the time and effort to teach me the art of scientific research and guide me safely towards the light at the end of the tunnel. Also, I would like to thank my two thesis readers Lt Col William Hobart and Major Paul Bailor and my sponsor Major Peter Reath for their time and effort. To all of you, I would like to say "your insights and comments made the research something I can be proud of."

Third, I would like to thank my sponsor organization WL/FIPA, Wright Laboratories, Flight Dynamics Directorate, Wright-Patterson AFB, OH, for providing the motivation and funding to make this research possible.

Finally, a special thanks to some of my classmates Steve, Rich, Doug, Cathy, KK, Mike, Rob, Jordan, Chuck, Al, Bill, John, Keith, Dan, and Dave all of who helped make the difficult times at AFIT bearable.

Table of Contents

List of Figures

List of Tables

Abstract

While the cyclic executive and fixed-priority scheduling strategies have been sufficient to handle traditional real-time requirements, they are insufficient for dealing with the complexities of next-generation real-time systems. New methods of intelligent control must be developed for guaranteeing on-time task completion for real-time systems that are faced with unpredictable and dynamically changing requirements. Implementing real-time processes as partial-solution tasks is one technique that may be beneficial. This type of task, when combined with intelligent control, has the potential for increasing pre-runtime schedulability, system maintainability, and runtime robustness. This research investigates the benefits of partial-solution tasks by experimentally measuring the change in performance of 11 simulated real-time systems when converted from all-or-nothing tasks to partial-solution tasks. Results from the experiments indicate that partial-solution tasks have the potential to decrease missed deadlines and increase a systems' average solution quality. The results also suggest that best performance gains can be achieved using *Optimistic* partial-solution tasks where the bulk of solution quality is achieved early during task execution. The framework used in this research was developed to measure the general case performance characteristics of partial-solution tasks. As a by-product, the research resulted in a framework that can also be used to measure specific case characteristics.

NEXT-GENERATION REAL-TIME SYSTEMS:

INVESTIGATING THE POTENTIAL OF PARTIAL-SOLUTION TASKS

I - Introduction

The relatively new field of Real-Time Artificial Intelligence (RTAI) marks the convergence of two areas of computing research: Artificial Intelligence (AI) and Real-Time Systems. The need for RTAI has developed because AI efforts are moving toward more realistic domains requiring real-time response, and real-time systems are moving toward more complex applications requiring intelligent behavior.

Traditional techniques for developing intelligent systems often result in unpredictable runtime performance and thus are unsuitable for incorporation into real-time systems. In turn, real-time systems often employ inflexible scheduling strategies that are incapable of handling the dynamic nature of intelligent systems.

To meet the requirements of next generation real-time systems, methods for dealing with dynamic and unpredictable environments need to be developed. Partial-solution tasks provide one method for maintaining system integrity under less than desirable conditions. The robustness and flexibility of these tasks can make it possible to meet strict deadline constraints in the face of limited resources and dynamically changing requirements.

1.1 Problem Statement

Real-time computer systems can be classified into three general categories: hard real-time, soft real-time and complex real-time:

- Hard real-time systems must produce computationally correct results while satisfying stringent timing constraints (deadlines). No value is added to a hard real-time system if its tasks fail to meet their deadlines. In some applications, failure can even result in catastrophic loss of life or property [Xu, 1991:10].

- Soft real-time systems differ from hard real-time systems in the extent to which failure of tasks to meet deadlines impacts the system. In soft real-time systems, the value added by a task's successful completion diminishes as latency increases.

- Complex real-time systems contain combinations of both hard real-time tasks and soft real-time tasks. Failure to meet deadlines in complex systems can result in a range of consequences from catastrophic to inconsequential.

Hard real-time systems often operate in well-defined environments, where task requirements and performance characteristics are deterministic in nature, thus allowing system designers to account for all possible resource combinations prior to implementation. Once recorded, these combinations are scheduled using well-founded static (pre-runtime) scheduling techniques. Next-generation RTAI systems on the other hand,

> ... exhibit a great deal of adaptability and complexity, making it impossible to precalculate all possible combinations of tasks that might occur. This precludes use of static scheduling policies common in today's real-time systems. We need new approaches for real-time scheduling in such systems, including on-line guarantees and incremental algorithms that produce better results as a function of available time [Stankovic, 88:10].

Rarely is it necessary for next-generation real-time systems to meet all their deadlines [Stankovic, 1993b]. While this is a desirable goal, the resource requirements for accomplishing it are often impractical and sometimes impossible [Whelan, 92]. Because of this, the current emphasis for next-generation real-time systems is on building intelligent runtime controllers that can adapt system performance by trading solution quality for time

The ultimate goal of the intelligent controller is to achieve 100 percent solution quality. A system quality of 100 percent means that the controller is able to schedule all resource requirements without missing a single deadline or timing constraint. However, due to the dynamic and unpredictable nature of next-generation real-time systems, there may not exist enough processing power to solve all problems simultaneously [Whelan, 1992 and Raeth, 1994]. Because of this, some tasks will be stopped before completion, others will be scheduled with limited resources, and still others will not be scheduled at all. Therefore, the actual goal of the intelligent controller is to generate schedules that optimize the total system performance by minimizing missed deadlines and maximizing the solution quality of individual tasks.

Using missed deadlines and solution quality as a discriminate, we can divide real-time tasks into two categories: all-or-nothing and partial-solution. All-or-nothing tasks achieve 100 percent solution quality when allowed to execute for their total required duration. Any execution time short of the required time results in a zero percent solution quality and a missed deadline. Partial-solution tasks, on the other hand, are capable of generating an acceptable partial quality solutions, even when faced with limited resources or stopped prior to completion. Such tasks appear to be better suited for manipulation by next-generation real-time systems that use intelligent control techniques.

1.2 Thesis Objective

This research investigates the benefits associated with developing a real-time system using partial-solution techniques. Due to time constraints, resource constraints, and a lack of domain-specific information, it focuses on three potential benefits: increased pre-runtime schedulability, increased pre-runtime maintainability, and increased runtime robustness. Furthermore, these characteristics are examined in an environment void of domain-specific knowledge. While domain knowledge is the key to intelligent scheduling, it is both difficult and expensive to gather. By examining the domain-independent properties of partial-solution tasks, it is sometimes possible to detect situations where system performance can be increased without the need for costly domain-specific information.

1.3 Approach

The approach used in this research is one of example and experimentation. Pre-runtime claims of increased schedulability and maintainability are supported using two examples. The first claim of increased schedulability is supported by converting a non-schedulable task set into a schedulable one by substituting partial-solution tasks for one or more all-or-nothing tasks. The second claim of increased maintainability is supported by allowing a search-based scheduler to automatically adapt the schedule generated in example one, in response to changing requirements.

Runtime claims of increased schedulability and robustness are investigated by experimentally measuring the effect of random sporadic task arrivals on several simulated real-time systems. During these experiments, the real-time systems are evaluated using only all-or-nothing tasks, only partial-solution tasks, and a mixture of all-or-nothing and partial-solution tasks. Results from these experiments are analyzed to determine if partial-solution techniques are more robust than all-or-nothing techniques.

1.4 Thesis Overview

This chapter provided a brief overview of the background, objectives, and approach used in developing this research. Following chapters address each of these subjects in more detail. In Chapter II, results of a literature review outline current issues and research applicable to the design and development of next-generation real-time systems. Chapter III draws upon the background information and provides a design methodology for comparing the performance characteristics of all-or-nothing and partial-solution tasks. Chapter IV describes the implementation of the support software used during the pre-runtime demonstration and the runtime experiments. In Chapter V, results from the experiment are presented and analyzed to determine if actual results reflect expected results. Finally, in Chapter VI, some conclusions are drawn about the actual benefits of partial-solution tasks and recommendations are made for follow-on work.

II - An Overview of Real-Time Intelligent Systems

2.1 Basic Requirements of Real-Time Systems

Real-time systems are those systems in which the correctness of the system depends not only on the logical results of computations, but also on the time at which the results are produced [Stankovic, 1993b]. Examples of some real-time systems include aircraft guidance and control, hospital monitoring and care, and nuclear power plant control. In general, real-time systems need to meet four basic requirements of timeliness, predictability, dependability, and simultaneousness [Hoogeboom, 1992:26-28].

- **Timeliness** - Traditionally, real-time tasks have been classified based on the degree to which failure to meet timing requirements affects the system. For hard tasks, timing violations can result in a disrupted control process, catastrophic failure, and even loss of life. For soft tasks, timing violations can cause undesirable effects, but do not result in fatal errors. A timely system must be able to meet all hard requirements while minimizing the impact of missed soft requirements.

- **Predictability** - Real-time systems should always be predictable. Given a set of inputs, the system's response to those inputs should be readily traceable from the system specification.

- **Dependability** - Real-time systems must be trustworthy. First, they must satisfy their specifications. Second, they must remain in a predictable state even when their runtime environment does not correspond to the specifications.

- **Simultaneousness** - Real-time systems must be able to meet timing requirements even when task resource requirements overlap.

2.2 Scheduling Techniques for Meeting Real-Time Requirements

Satisfying the timing requirements of real-time systems demands the scheduling of system resources according to some well-understood algorithms so that the timing behavior of the system is understandable, predictable, and maintainable. It is important to note that real-time system scheduling differs from scheduling problems usually addressed in operations research. In most operations research problems, the goal is to find an optimal static schedule that minimizes the response time for a given set of tasks [Stankovic, 1988:13]. For real-time systems, the ultimate goal is to find a static schedule that meets all resource requirements and timing deadlines.

Scheduling strategies for real-time systems can be classified as one of two types: *pre-runtime* or *runtime*. In pre-runtime scheduling, also known as off-line scheduling or static scheduling, the schedule for processes is computed off-line and requires that the major characteristics of the processes within the system be known in advance. In runtime scheduling, also known as on-line scheduling or dynamic scheduling, the schedule for processes is computed on-line as processes arrive.

The perceived advantages of pre-runtime scheduling include significant reductions in the amount of runtime resources required for scheduling and context switching and a strong guarantee of satisfying deadlines. The perceived disadvantage is an inflexibility to adapt to changing or unpredictable events.

The perceived advantages of runtime scheduling over pre-runtime scheduling include: no necessity to know the major characteristics of the system in advance, flexibility, and easy adaptabilty to changes in the environment. There are two major disadvantages with the runtime approach for satisfying timing constraints in a hard real-time system:

> The first problem stems from the basic assumption that the scheduler does not have any knowledge about the major characteristics of processes that have not yet arrived in the system. If the scheduler does not have such knowledge, then it is impossible to guarantee that all timing constraints will be satisfied, because no matter how clever the scheduling algorithm is, there is always the possibility that a newly arrived process possesses characteristics that make the process either miss its own deadline, or cause other processes to miss their deadlines.
>
> The second problem is the fact that the amount of time available for a runtime scheduler to compute schedules on-line is severely restricted, and the time complexity of most scheduling problems is very high. There are too many different possible combinations of times and orders in which processes may arrive and request use of system resources for the scheduler to compute at runtime, especially in complex systems where there are a large number of processes and resources. [Xu, 1991:134]

However,

> ... the robustness of modern and next-generation real-time AI systems, especially in such decision support arenas as combat aircraft cockpits, require a similarly robust dynamic task scheduling method. While current scheduling technology may view this as an intractable dichot only, it is still an issue that must be addressed. We can not let the difficulty of the problem cause us to ignore it. It's solution is too urgent [Raeth, 1994].

Because of the disadvantages, the majority of real-time systems have been developed using pre-runtime scheduling. Furthermore, they have been developed using one of two scheduling techniques: *cyclic executive scheduling* or *fixed-priority scheduling*.

2.2.1 Cyclic Executive Scheduling.

A cyclic executive is a control structure for explicitly interleaving the execution of real-time processes. The interleaving is done in a deterministic fashion so that execution timing is predictable. The process interleaving is non-preemptive and is defined using a pre-runtime *cyclic schedule.* The cyclic schedule describes a sequence of actions that need to be performed during a fixed period of time. The cyclic executive plays the cyclic schedule in an endless loop, repeating the schedule tasks until told otherwise. The execution of a real-time system can be divided into one or more cyclic schedules that correspond to different modes of operation. The cyclic executive switches between cyclic schedules in response to real-time events.

The basic building block of the cyclic executive is the *periodic process.* A periodic process consists of an action repeated at regular intervals. For scheduling purposes, a periodic process can be defined as a triple (c, d, p) , where c is the process computation time (duration), d is the process deadline, and p is the process period. The length of the cyclic schedule is equal to the lowest common multiple of the periods of its constituent processes.

There are three advantages in using a cyclic executive for structuring a real-time application. First, cyclic executives make it possible to predict the entire future history of the system. Since the entire execution schedule is predetermined, it is clear that all response requirements can be met. Second, because no task can ever be interrupted during its execution, there is no requirement for application preemption. Thus overhead can be kept low. Finally, because of determinism in the schedule, it is possible for each periodic task to exhibit a very low deviation in the size of the interval between activation and completion [Locke, 1992:6]. A characteristic which is commonly referred to as low jitter.

There are two basic disadvantages in using a cyclic executive. The most serious disadvantage is the application fragility. Most cyclic schedules are carefully hand-crafted to meet a specific set of timing requirements. Generally, they do not react well to change. Introducing new functionality can often result in timing requirements that exceed the capabilities of the original cyclic schedule. Another problem is in the cyclic executive's handling of processes with execution times which are long compared to the period of the highest rate cyclic task. The most common solution to this problem is to arbitrarily break the long task into multiple short tasks by manually inserting preemption points [Locke, 1992:7]. While this solves the problem, it also adds to the fragility of the system.

2.2.2 Fixed-Priority Scheduling.

A fixed-priority executive is a control structure for interleaving the execution of real-time processes. Unlike the cyclic executive, it does not use a predetermined schedule to decide which process to execute. Instead, it makes the decision at runtime based on the relative importance of processes as specified by a set of predetermined (fixed) priorities. Prior to the early seventies, our ability to characterize the predictability of fixed-priority systems was weak [Locke, 1992:46]. In 1973, however, Liu and Layland published a paper that described the use of *rate monotonic scheduling* as a paradigm for designing fully predictable hard real-time (fixed-priority) systems [Liu, 1973].

Rate monotonic scheduling is simply the use of a preemptable, fixed-priority executive to execute a set of periodic processes whose priorities are ordered monotonically increasing with respect to the frequency of the process. Liu and Layland proved in their paper that assigning priorities in this manner results in optimal schedules for the class of all fixed-priority scheduling algorithms.

There are several advantages in using fixed-priority scheduling. The principal advantage is the predictability of the entire process set when faced with changing requirements. Since the predictability is based on knowledge of total processor utilization, determining schedulability for additional processes only requires calculating the maximum allowed processor utilization, and comparing it to the actual processor utilization of the new system. If the maximum utilization is not exceeded, then the process set is schedulable and process priorities can be assigned. Another advantage of the rate monotonic approach is that it normally does not require system designers to decompose processes at arbitrary points to fit within system timing requirements. Rather, it relies on the runtime executive to preempt the process wherever and whenever required. The final advantage of rate monotonic scheduling is its degree of stability. Simply exceeding an allocated utilization does not always result in failure of a process or the system. As long as the total system utilization remains below the maximum scheduling bound, the system will continue to meet its timing constraints.

The principal disadvantage of fixed-priority scheduling is its inability to map the many different execution orderings of processes onto a rigid hierarchy of priorities. Another problem with fixed-priority scheduling is that it is only able to produce a limited subset of possible schedules. This severely restricts its ability to satisfy timing and resource constraints at runtime [Xu, 1991:137]. Finally, fixed-priority scheduling does not define a tight timing constraint on the actual completion time of a task, other than ensuring that it completes prior to the end of its period. Because of this characteristic, there can be significant jitter generated for processes with long periods.

2.2.3 Scheduling Sporadic Tasks. Both the cyclic executive and fixed-priority scheduling strategies were developed around the periodic process. Another type of process that is becoming more prevalent in real-time systems is the *sporadic process*. A sporadic process, also known as a dynamic process or an asynchronous process, exhibits non-periodic arrival times. To schedule sporadic processes using a cyclic executive or fixed-priority scheduling strategy, designers are required to translate the sporadic process into an equivalent periodic process. The period of the new process is calculated from the worst case inter-arrival time of consecutive sporadic processes. Two problems exist with translating sporadic processes into periodic processes.

First, sporadic translation can result in extremely inefficient schedules. Periodic resources, pre-allocated for the sporadic process, use valuable system resources even when no sporadic processes actually arrive. This increases the amount of system resources that must be made available. In this era of demands for affordable systems, it is necessary to find ways to decrease the need for system resources. This can somtimes be acheived using more complex scheduling techniques.

Second, the translation process does not guarantee that all sporadic processes will meet their timing constraints. It is always possible that the actual worst case inter-arrival time of the sporadic processes will exceed the value used to generate the periodic server.

2.2.4 Dynamic Deadline Driven Scheduling. Another method for handling periodic and sporadic processes is a runtime strategy called *deadline driven scheduling*. Using this method, priorities are assigned to processes according to their deadlines: the closer the deadline, the higher the priority. At any instant, the process with the highest priority (shortest deadline) will be executed. Liu and Layland proved that deadline driven scheduling can result in an optimal schedule when presented with a process set that is schedulable using some fixed-priority assignment [Liu, 1973:61].

Deadline driven scheduling has two advantages over fixed-priority scheduling. First, it can increase system utilization bounds from 69 percent to 100 percent. Secondly, it does not waste resources on sporadic events that do not occur. Generally speaking, a combination of fixed-priority scheduling or cyclic executive scheduling with deadline driven sporadic scheduling appears to produce the most benefits [Liu, 1973:61].

2.3 Next Generation Real-Time Systems

Three major forces are pushing real-time systems into the next generation: movement of real-time systems toward more complex applications requiring intelligent behavior, movement of artificial intelligent systems toward more realistic domains requiring real-time response, and rapid advances in hardware. Next-generation, critical, real-time systems will require greater flexibility, dependability, and predictability than is commonly found in today's systems.

The *DARPA Neural Network Study* notes that there is a general trend toward increasing problem complexity and specificity, as well as an increasing need for task-independent and autonomous software [Gschwendtner, 1988:262]. Current practices for guaranteeing real-time performance are inadequate to handle the increased complexity and dynamic requirements of next-generation real-time systems. New methods need to be developed to produce predictable behavior while providing enough flexibility to deal with failures, non-deterministic environments, and system evolution.

One method currently under investigation throughout both the real-time and AI communities is *intelligent real-time control*. Intelligent real-time control can be defined as a computer control system that can adapt its scheduling policies to meet changing operational requirements. Typically, such a system would use both pre-runtime and runtime scheduling strategies to guarantee real-time requirements. Pre-runtime scheduling would be used to guarantee the response times of some set of critical processes, while runtime scheduling would be used to trade the solution quality of less important tasks for time, to satisfy the timing constraints of more important tasks. Figure 2.1 illustrates the components of a next-generation real-time system with intelligent control.

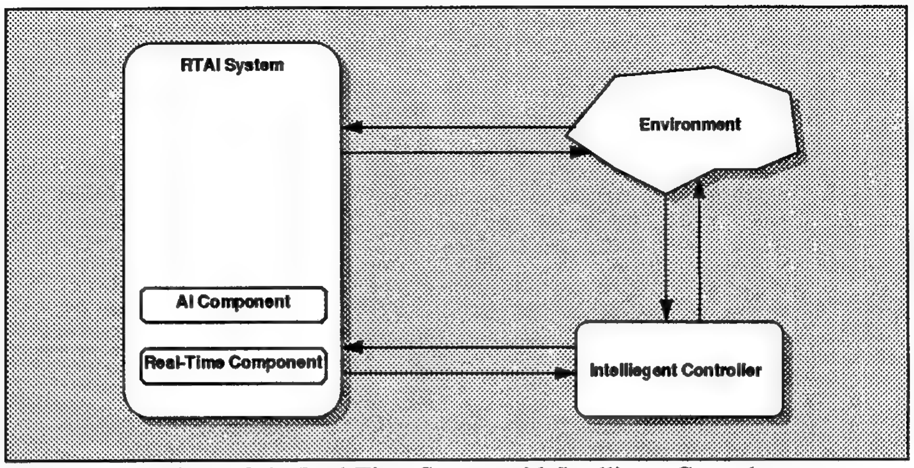


Figure 2.1: Real-Time System with Intelligent Control

Two research architectures, the *Spring Architecture* and the *Cooperative Intelligent Real Time Control Architecture*, have been addressing the issue of intelligent real-time control.

2.3.1 Spring Architecture. The Spring Architecture is an intelligent real-time control architecture being developed by the Department of Computer and Information Science at the University of Massachusetts. Spring is being developed to challenge several basic assumptions upon which traditional real-time systems are built. It advocates a new paradigm for providing on-line dynamic guarantees to certain types of processes.

Spring's new paradigm is based on the concept of *reflection*. Reflection is defined as the process of reasoning about and acting upon the system itself. Part of this action may be altering the system's own structure from within. Traditionally, real-time systems perform computation acts to monitor sensors, perform calculations, and control actuators. If, in addition, there is computation about the monitoring processes, quality of calculations, and dependability of actuators, then there is meta-level structures and reflective possibilities. Proper use of reflection can result in systems with increased flexibility, understandabilty, analyzability, and dependability [Stankovic, 1993b:2].

Reflection in the Spring Architecture is provided via a multi-level, multi-dimensional scheduling strategy [Stankovic, 1993b:6]. To achieve the multi-level design, tasks are divided into three categories:

- **critical** - Critical tasks must always make their deadlines, otherwise catastrophic system failure may occur.

- **essential** - Essential tasks are necessary to the operation of the system, have specific timing constraints, and will degrade system performance if timing constraints are not met. If an essential task misses a deadline, it will not cause catastrophic failure.

- **non-essential** - Non-essential tasks may or may not have timing constraints. Failure to meet non-essential timing constraints results in little or no loss in system performance. Background tasks, long range planning tasks, and maintenance functions fall into this category.

The multi-dimensional scheduling strategy provides different guarantees for each task category. Critical tasks are provided the greatest guarantee by using *a priori* knowledge and pre-runtime scheduling techniques that pre-allocate system resources. Due to the large number of essential tasks and to the extremely large number of task combinations, it is not possible, nor desirable to reserve resources for essential tasks. Rather, these tasks are provided a lesser guarantee through a dynamic runtime planning scheduler that provides acceptable performance based on available resources. In some circumstances, it is necessary to pre-allocate resources for essential tasks. These tasks have extremely tight deadlines and cannot meet timing constraints if scheduled using the planning scheduler. Non-essential tasks are executed only when they do not affect critical or essential tasks.

On-line guarantees for essential tasks have a very specific meaning within the Spring Architecture:

> ... it allows the unique abstraction that at any point in time the operating system knows exactly which tasks have been guaranteed to make their deadlines, what, where and when spare resources exist or will exist, a complete schedule for the guaranteed tasks, and which tasks are running under non-guaranteed assumptions. However, because of the non-deterministic environment, the capabilities of the system may change over time, so the on-line guarantee for essential tasks is an *instantaneous* guarantee that refers to the current state. Consequently, at any point in time we have a *macroscopic* view that all critical tasks will make their deadlines and we know exactly which essential tasks will make their deadlines given the current load [Stankovic, 1993a:9].

The main advantage of the Spring approach is it separates deadlines from importance. Critical tasks are of the utmost importance and are guaranteed (within reasonable constraints) for the lifetime of the system. Essential tasks are assigned a level of importance that varies in response to changing system requirements. The planning scheduler combines the importance level of each essential task with *a priori* knowledge to generate a schedule that maximizes system performance with respect to current needs.

2.3.2 Cooperative Intelligent Real Time Control Architecture. The Cooperative Intelligent Real Time Control Architecture (CIRTCA) is an intelligent real-time control architecture developed by David Musliner to support his doctoral work at the University of Michigan [Musliner, 1993].

Musliner perceives an apparent conflict between the nature of AI and the needs of real-world, real-time control systems. Traditional real-time systems operate in well-defined environments; AI systems do not. Most AI systems have been developed without much attention to the resource constraints that motivate real-time research. Many of these AI systems attempt to approximate intelligent reasoning by using search techniques that are slow, costly, and unpredictable. To achieve predictable intelligent real-time control, researchers have focused either on restricted AI techniques or reactive systems that retain little of the power of traditional AI.

CIRTCA is designed to retain the power of traditional AI while guaranteeing hard real-time requirements. To accomplish this, it uses an AI subsystem (AIS) to reason about task-level problems that require powerful but unpredictable reasoning methods, while a separate real-time subsystem (RTS) uses its predictable performance characteristics to deal with control-level problems that require guaranteed response times. Figure 2.2 illustrates the separation of components in the CIRTCA design.

The basic building block of the CIRTCA design is the test-action-pairs or TAPs. A TAP consists of a fixed set of tests (or preconditions), a set of actions, worst-case timing data, and a list of resource requirements. Multiple TAPs are combined at runtime to form a cyclic schedule that is used by the RTS.

The AIS runs asynchronously with the RTS and reasons about task-level goals in hopes of generating an optimal set of TAPs. The AIS is designed to sacrifice completeness at the task-level in order to achieve predictability at the TAPs or control-level. The AIS passes a TAPs list to the scheduler, which builds a cyclic schedule by reasoning about the maximum period of the TAPs, their worst case execution times and resource needs, and the resources available from the RTS. If the scheduler is unable to build a successful schedule, it reports a failure to the AIS, which must adjust the TAPs list to relax scheduling constraints.

When the scheduler passes a guaranteed schedule to the RTS, it also passes a list of unguaranteed TAPs. The RTS runs through the guaranteed TAP schedule, checking the tests for each TAP and firing those TAPs whose tests return true. If a TAP test returns false, the RTS uses the time schedule for the guaranteed TAP to search for and invoke one or more of the unguaranteed TAPs. Transition from one schedule to another is accomplished when the RTS encounters a **check-for-new-schedule** TAP that returns true.

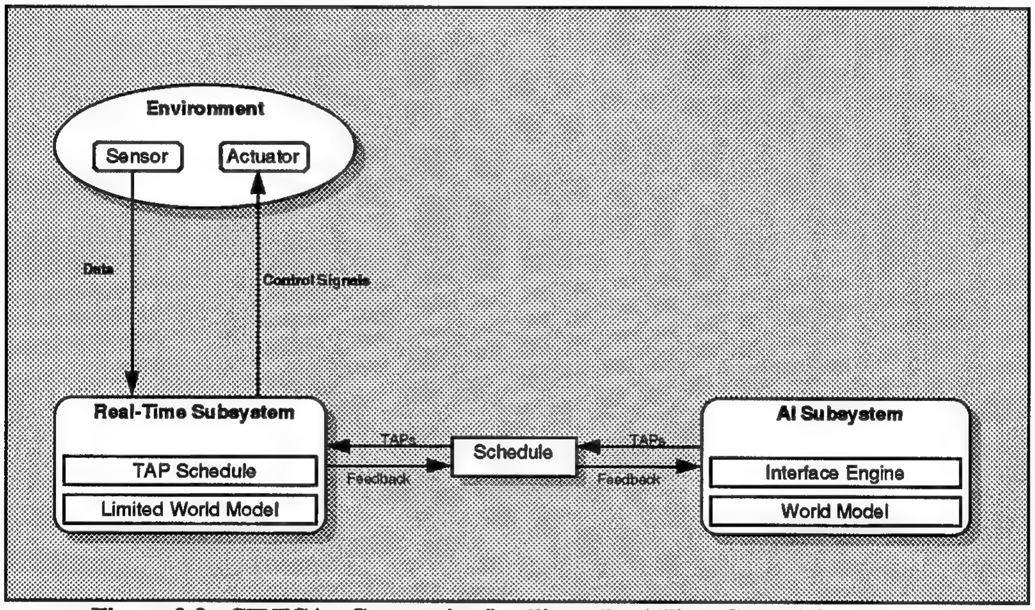


Figure 2.2: CIRTCA - Cooperative Intelligent Real-Time Control Architecture

A basic assumption of the CIRTCA design is that the system can remain in a safe set of states for an indeterminate amount of time without violating its control-level goals. All changes to RTS must be made via the AIS. At no time is the cyclic schedule interrupted or a TAPs execution preempted. If interruptions are required, then they must be scheduled in advance by the AIS and handled in a polled manner. A drawback of this design requirement is that it can lead to degraded or inappropriate performance, if the AIS is slow in generating new schedules.

David Musliner identifies the primary advantage of the CIRTCA design as the asynchronous operation of both the AIS and the RTS. By operating in this fashion, the AIS need not conform to the rigid performance restrictions of the RTS and can bring to bear the full power of traditional AI techniques.

However, by requiring the AIS to handle all scheduling requirements, CIRTCA has bound the effectiveness of the real-time subsystem to the response time of the AI subsystem. This characteristic is precisely the reason why real-time AI designers have been forced to use restrictive or reactive techniques in the first place.

2.4 Partial-Solution Tasks and Next Generation Systems

When combined in a next-generation real-time system, partial-solution tasks and intelligent control interact jointly to enhance system performance. Partial-solution tasks enhance performance by providing the intelligent controller with more opportunities than would be available using all-or-nothing techniques. The intelligent controller enhances the system performance by determining which partial-solution tasks to degrade in order to maximize potential benefits. For example, consider the task schedule presented in Figure 2.3.

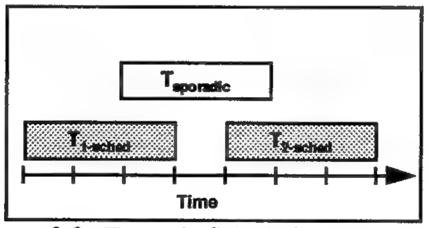


Figure 2.3: Example Scheduling Requirement

Tasks T_1 and T_2 represent static tasks that have been scheduled using pre-runtime scheduling techniques. Task $T_{sporadic}$ represents a sporadic task that must be scheduled using a runtime dynamic scheduler. If the processes are implemented as all-or-nothing tasks then only two situations can be considered. In the first case the sporadic task can be ignored, resulting in 100% benefit from the static tasks but 0% benefit from the sporadic task. In the second case the sporadic task can be scheduled, resulting in 100% benefit from the sporadic task but 0% benefit from static tasks. (See Figure 2.4.)

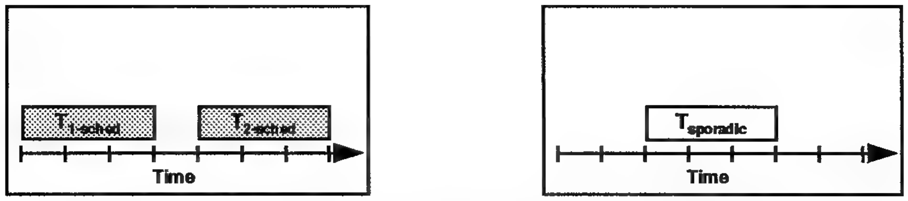


Figure 2.4: All-Or-Nothing Solutions to Example Scheduling Requirement

By implementing the processes as partial-solution tasks the number of solutions is increased from two to six. The four additional solutions, shown in Figure 2.5, allow the scheduler to schedule all three tasks at

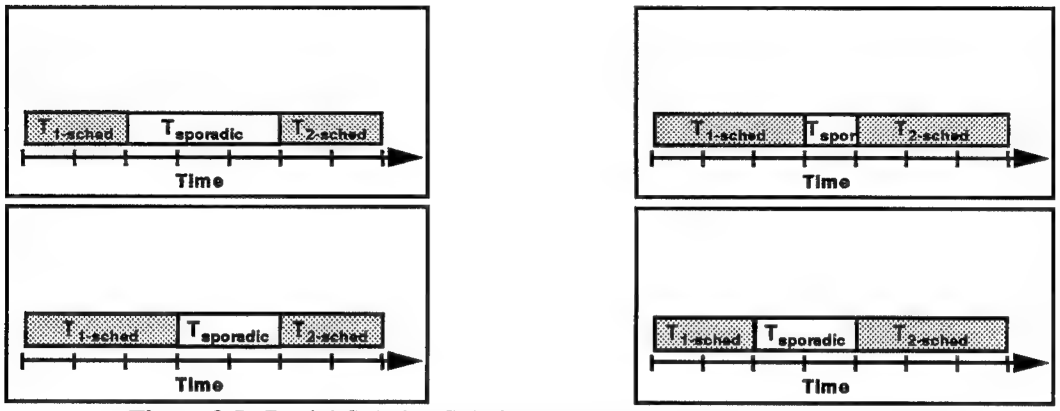


Figure 2.5: Partial-Solution Solutions to Example Scheduling Requirement

less than 100% benefit. This capability results in a decrease in the number of missed deadlines and introduces the possibility of increased total benefit to the system. The additional solutions also increase the scheduling complexity and require the use of a domain dependent intelligent controller to determine the optimal solution.

The benefits shown in the example above also apply to pre-runtime scheduling. It is possible to use the same process during pre-runtime scheduling if the requirements of $T_{sporadic}$ are known in advance. In that situation, partial-solution tasks make it possible to generate a feasible static schedule for all tasks, when none would exist using all-or-nothing techniques. Further benefits can also be envisioned during software maintenance. In this situation, $T_{sporadic}$ could represent an additional requirement that was not considered during system development. Adding a new all-or-nothing task to the system could require the procurement of an additional processor. Adding a new partial-solution task would degrade system performance, but allow the system to meet its requirements without the need for additional hardware.

There are some drawbacks to using partial-solution techniques. First, it is highly unlikely that all processes can be implemented as partial-solution tasks. In fact, in some systems, there may be no process that can be implemented as a partial-solution task. Second, the increased number of opportunities will also increase the complexity of the runtime scheduling process. As a consequence, it may take the intelligent controller longer to generate the solution, since more schedules must be evaluated. Finally, there may be some cost associated with the conversion of all-or-nothing tasks to partial-solution tasks. These costs could include increased execution time, increased memory requirements, and increased software complexity. However, even with the drawbacks, there is still a compelling need to evaluate the capabilities of partial-solution tasks.

2.5 Summary

This chapter has described the difficulties associated with guaranteeing the four basic requirements of mission-critical real-time systems. While the cyclic executive and fixed-priority scheduling strategies have been sufficient to handle traditional real-time requirements, they are insufficient for dealing with the complexities of next-generation real-time systems. New methods of intelligent control must be developed for guaranteeing real-time systems when faced with unpredictable and dynamically changing requirements. Implementing real-time processes as partial-solution tasks is one technique that may be beneficial in the development of next-generation systems that use intelligent control. In the following chapter a design methodology is presented for exploring some of the potential benefits associated with partial-solution tasks.

III - Design Approach, Assumptions, and Key Decisions

Chapter II provided some background information on the design and implementation of next-generation real-time systems. This chapter draws upon that information and provides a high-level design for implementing a system with the specific objective of comparing the performance characteristics of partial-solution and all-or-nothing tasks. Following the high-level specification, some basic design assumptions are stated and several possible design decisions are discussed.

3.1 High-Level Research Objective

The primary hypothesis presented in this research is that partial-solution tasks have the potential for increasing the schedulability, maintainability and robustness of next generation real-time systems. To defend this hypothesis, this research shows that partial-solution tasks can:

- increase the pre-runtime schedulability of some real-time systems by generating feasible schedules that could not be produced with all-or-nothing tasks,
- increase the maintainability of the real-time system by automatically adapting pre-runtime schedules to scheduling constraints rather than hand-crafting schedules and tasks each time requirements change, and
- increase the runtime schedulability and robustness of the real-time system by retaining some profit from tasks that are unable to meet their full processing requirements due to preemption by more important tasks.

It is beyond the scope of this research effort to generate a mathematical proof supporting the claims present in the paragraph above. Instead, this thesis demonstrates the potential benefits of partial-solution tasks through example and experimentation.

3.2 Pre-Runtime Claims

It can be argued that the pre-runtime claims of increased schedulability and maintainability are intuitive. The very design of partial-solution tasks makes it possible for them to produce results even when faced with limited resources. However, it is important that this research demonstrates a practical example of these claims since they provide the foundation on which less intuitive runtime claims are based.

The first claim of increased schedulability is supported using a single example that converts a non-schedulable task set into a schedulable one by substituting partial-solution tasks for one or more all-or-nothing tasks. The second claim of maintainability is supported by allowing a search-based scheduler to automatically adapt the schedule generated in example one, in response to changing requirements. Both examples are conducted using *theorem 1* of the rate-monotonic scheduling algorithm and a set of tasking requirements known as the Generic Avionics Platform.

3.2.1 Rate Monotonic Scheduling Algorithm.

The rate monotonic scheduling algorithm (RMA) was selected as the pre-runtime scheduling strategy for two reasons. First, rate-monotonic scheduling provides a simple mathematical technique for verifying the schedulability of periodic, aperiodic, and synchronous tasks. Secondly, there is an abundance of literature describing its application to hard real-time systems.

Theorem 1 of the RMA guarantees that in the worst case conditions, all deadlines will be met if the processor utilization of the periodic tasks does not exceed a predetermined bound. Processor utilization can be calculated by summing the execution times of each task divided by the respective period of each task. This function is shown in equation 3.1. 69.3 percent is the lower bound on feasible processor utilization.

$$X = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) \qquad (3.1)$$

where

X is the processor load
Ci is the execution time of process *i*
Ti is the period of process *i*
n is the number of processes

The upper bound is 100 percent, which decreases monotonically to 69.3 percent as the number of tasks approaches infinity.

As long as the processor load is below the maximum utilization bound, the task set is always schedulable. Additional theorems have been developed that can guarantee schedulability for task sets with utilization levels higher than the limit presented in *theorem 1* [Liu, 1973]. However, these theorems are not required to support the claim of increased schedulability provided by partial-solution tasks.

3.2.2 The Generic Avionics Platform.

The Generic Avionics Platform (GAP) was developed by personnel from IBM, the Naval Weapons Center, and the Software Engineering Institute to determine how to use the Ada language in a hard real-time system. It presents a set of generic tasks expected in modern fighter weapon systems [Locke, 1990:118]. These tasks, which are outlined in Table 3.1, represent a set of non-dynamic scheduling requirements that is non-schedulable using *theorem 1* of the rate-monotonic scheduling algorithm, since their total processor utilization exceeds the maximum bound for 13 tasks of 0.711.

Table 3.1: GAP Timing Requirements

GAP Timing Requirements				
Task Name	Period	Duration	Utilization	Total Utilization
Contact Mgnt	0.025	0.005	0.200	0.200
Tracking Filter	0.025	0.002	0.080	0.280
Target Update I	0.050	0.005	0.100	0.380
Nav Update	0.059	0.008	0.136	0.516
Hook Update	0.065	0.002	0.031	0.547
Graphic Display	0.080	0.009	0.113	0.660
Target Update II	0.100	0.005	0.050	0.710
Status Update I	0.200	0.003	0.015	0.725
Steering Cmds	0.200	0.003	0.015	0.740
Stores Update	0.200	0.001	0.005	0.745
Keyset	0.200	0.001	0.005	0.750
Nav Status	1.000	0.001	0.001	0.751
Status Update II	1.000	0.001	0.001	0.752

The GAP was selected for two reasons. First, it is a small set of requirements that can be easily handled by *theorem 1* of the rate monotonic scheduling algorithm. Secondly, it closely resembles the type of system being developed by the sponsor of this research. While more realistic timing requirements would have been desirable, they are currently not available from the sponsor.

3.3 Runtime Claims

Supporting the runtime claim of robustness is not as simple as supporting the two pre-runtime claims of schedulability and maintainability. The dynamic nature of next-generation real-time systems make it impossible to determine *a priori* the actual runtime characteristics of a system. Generating a single example that proves a system meets all its deadlines would require a simulation that generates all possible combinations of periodic and sporadic tasks. Therefore, this thesis will not try to prove that partial-solution tasks are more robust than all-or-nothing tasks. Rather, it will show a general trend of increased robustness by experimentally measuring the effect of random sporadic task arrivals on 11 simulated real-time systems.

One of the simulated real-time systems consists of a cyclic schedule generated from the GAP timing requirements described in Table 3.1. The other 10 systems consist of the cyclic schedules described in Appendix A. The duration and importance for these schedules were generated using a pseudo-random number generator. The random number generator was` also used to generate importance values for GAP tasks.

3.3.1 Randomly Generated Task Data. The decision to use randomly generated schedules and randomly generated sporadic tasks is necessitated by the nature of next-generation real-time systems. Each system is composed of a set of tasks that have varying computational requirements, predictability requirements, and timing constraints. Because of this, it is impossible to develop a single set of tasks that represents all real-time systems. Statistically or randomly generating periodic and sporadic tasks can produce false or misleading results. However, it can also generate meaningful results when no additional domain-specific information is available.

3.3.2 Performance Measures. Before proceeding, it is important to discuss the issue of performance measures. Michael Whelan describes four performance measures for real-time systems: speed, responsiveness, timeliness, and graceful adaptation [Whelan, 1992].

- Speed. "This performance measure refers to the number of tasks executed per unit time. Speed is highly dependent upon the processing hardware. Generally, more and faster processors increase this performance measure" [Whelan, 1992].

- Responsiveness. "Responsiveness refers to the ability of a system to take on new tasks quickly. Operating in a rapidly changing environment, a responsive system perceives new developments early enough to compose and execute responses, possibly at the expense of ongoing tasks that may be delayed or even abandoned" [Dodhiawala, 1988].

- Timeliness. "This measure characterizes the system's ability to conform to task priorities. Assuming that not all tasks can be finished by their deadlines, a timely system is one that finishes as many as possible" [Whelan, 1992].

- Graceful Adaptation. "This refers to the ability of the system to reset task priorities according to changes in the resource availability and/or demand and workload" [Shamsudin, 1991].

Speed, responsiveness, and timeliness are not used in this research since they are primarily indicators of processing power and scheduler capabilities. Graceful adaptation, on the other hand, indicates the robustness of a system by measuring a change in system performance due to increased or unexpected resource requirements. Generally, a system that exhibits only a small decrease in system performance under increased workload will be considered more robust than one that exhibits a large change under the same conditions.

It is difficult to accurately determine a single metric for measuring the graceful adaptation of a real-time system. The separation of tasks into different levels of importance requires the scheduler to pass judgment on the relative benefits of each task. For example, it is difficult to determine which is better, executing a single high-importance task of long duration, or executing several medium-importance tasks of short duration. The selection of an appropriate metric may vary drastically depending on domain-specific requirements. This research proposes four generic metrics for measuring graceful adaptation in terms of average system quality. These four metrics are: percent of missed deadlines for periodic tasks, percent of missed deadlines for sporadic tasks, average solution quality for periodic tasks, and average solution quality for sporadic tasks.

3.3.3 Cyclic Executive.

With the performance measures identified, the design effort must now focus on developing a runtime system for gathering performance data. Using the Spring Architecture and CIRTCA as models, we can determine that this system should include a pre-runtime scheduler, a runtime scheduler, and a dynamic task generator.

The cyclic executive method of controlling runtime tasks has been selected for this phase of the demonstration because it provides tighter control over the start and stop times of individual tasks than can be achieved using a rate-monotonic approach. Fixed-priority schedules, while easy to create using the rate-monotonic scheduling algorithm, do not allow accurate control over task execution. Just because a task has a high priority (based on its short period) does not mean that it should always preempt more important tasks with long periods. Furthermore, the knowledge representation used in the cyclic schedule can provide a runtime scheduler with significantly more information than is available using a single fixed-priority.

Conversion of the GAP fixed-priority schedule to a cyclic schedule is a simple two-step process. First, the length of the major cycle is found by calculating the least common multiple of all the task periods. Second, the cyclic schedule is generated by simulating the actions of the fixed-priority scheduler for a period equal to the major cycle. The resulting product is a timeline that specifies the start and stop times of each task. It is unnecessary to convert the random schedules since they are automatically generated in a cyclic format. Descriptions of the random cyclic schedules and GAP cyclic schedule are provided in Appendix A.

A reactive, non-intelligent, importance-based scheduler has been selected for scheduling the sporadic tasks activated by the dynamic task generator. This method was selected over a planning scheduler because it decreases the dependencies between the scheduler's capabilities and the system performance. Increased dependencies could make it difficult to determine if increased robustness is due to the partial-solution tasks or a result of some knowledge embedded in the planning scheduler. Additionally, because the importance, duration, and response times of sporadic tasks will be generated randomly, no domain information will be available to support a planning scheduler.

3.4 Design Decisions

Based on sponsor requirements, scoping requirements, background research, and software and hardware constraints it was necessary to make several key design decisions. This section lists these decisions and describes the reasons they were made.

A direct result of sponsor requirements was the decision to implement the runtime experiment in Ada. This requirement was levied by the sponsor to comply with Public Law 101-511 which mandates "where cost effective, all Department of Defense software shall be written in the programming language Ada." The languages currently used by the sponsor include C and C++. While special exemptions can be granted by the Secretary of Defense, the number of exemptions approved each year is decreasing. By using the Ada language, this research provides essential feedback that can be used to expedite the conversion of existing software to Ada or to justify continued waiver requests.

Another decision partially influenced by the sponsor was the selection of the Verdix Multi-Processor Ada (MP Ada). Early in the research, it became apparent that an Ada compiler for developing next-generation real-time systems should provide certain capabilities. Some of these capabilities include: preemptive priority scheduling, dynamic task assignment, task suspension and resumption, task-to-processor assignment, interrupt handling, and absolute time control. While most Ada compilers available at AFIT do not provide these capabilities, Verdix MP-Ada, which is available to the sponsor and to AFIT, does provide these capabilities via a nonstandard extension to the Ada language.

A consequence of selecting Verdix MP-Ada is that both available copies exist on Silicon Graphics computers. A problem with the Silicon Graphics is its available clock resolution. According to system documentation the system has a timer resolution of approximately 0.01 seconds. Sometimes, this resolution is an order of magnitude larger than the timing requirements identified in the GAP. To compensate for this problem, the duration and period of GAP tasks have been modified to be multiples of the clock resolution. In addition to this, several tasks have been omitted and others modified to reduce the length of the cyclic schedule. The primary reason for doing this was to reduce the amount of time required for each experiment. The results of these modifications are detailed in Table 3.2.

Another problem with using the Silicon Graphics is its use of the UNIX operating system. While UNIX is a very powerful general purpose operating systems, it lacks the rigorous process control required for regulating hard real-time tasks [Musliner, 1993]. As a result, there may be a degree of uncertainty injected into experimental data. An effort is made to minimize this uncertainty by limiting access to the machine and using the UNIX command *nice* to maximize the priority of the experimental system.

3.5 Design Assumptions

The primary assumption made by this thesis is that some elements of a next-generation real-time system can be implemented using partial-solution tasks. This assumption, which has been approved by the sponsor, can be supported by applying our definition of a partial-solution task to several example problems.

Table 3.2: Modified GAP Timing Requirements

GAP Timing Requirements						
Task Name	Period	Duration	ID	Cyclic Priority	Cyclic Period	Cyclic Duration
Contact Mgnt	0.025	0.005	1	ESSENTIAL	0.200	0.050
Tracking Filter	0.025	0.002	-	-	-	-
Target Update I	0.050	0.005	2	NONESSENTIAL	0.500	0.050
Nav Update	0.059	0.008	3	NONESSENTIAL	0.500	0.080
Hook Update	0.065	0.002	-	-	-	-
Graphic Display	0.080	0.009	4	CRITICAL	0.800	0.090
Target Update II	0.100	0.005	5	CRITICAL	1.000	0.050
Keyset	0.200	0.001	6	CRITICAL	2.000	0.010
Steering Cmds	0.200	0.003	7	NONESSENTIAL	2.000	0.030
Stores Update	0.200	0.001	8	NONESSENTIAL	2.000	0.010
Status Update I	0.200	0.003	9	NONESSENTIAL	2.000	0.030
Nav Status	1.000	0.001	-	-	-	-
Status Update II	1.000	0.001	-	-	-	-

In Chapter II, a partial-solution task was defined as a task that is capable of generating an acceptable partial solution under less than optimal timing constraints. One example of such a process is a technique used in graphics for guaranteeing image refresh rates. Most graphical simulations trade image quality for time. The more time that is available, the more accurate the image. However, to create believable animation, it is necessary to generate at least 15 graphical frames a second. If an image becomes too complex, it may be impossible to generate the entire image in real-time while still meeting the 15 frame requirement. To overcome this problem, graphical systems can use an iterative method that renders images starting with the closest object first and moving to more distant objects as time permits. Each iteration produces an acceptable partial solution and moves on only if it can continue to guarantee the required frame rate.

Another example is a multi-window display that organizes and updates its windows in an iterative or hierarchical order. Using the iterative approach, the device will display each window in a predetermined order. If time runs out, the task will complete the current window and suspend until more time becomes available, at which time it will continue from where it left off. The net effect is a partial solution quality that stretches out the window refresh period. Using the hierarchical approach, the device would display each window starting with the most important window first. If time runs out, the task will complete the current window and suspend until its next activation, at which time it will start again with the most important window. This approach produces a partial solution that updates only the most important windows in the available time.

Yet another partial-solution task is rule-based evaluation in expert systems. Rules can be ranked in priority order so the low-importance rules can be left out if time available is insufficient. Another possibility to consider is neural nets and the limitation of node summations. The extent of a search technique is another.

Each of these examples is a valid application of partial-solution tasks and shows their applicability to next-generation real-time systems. While not all processes can be developed as partial-solution tasks, it is reasonable to assume that some can be developed in this manner.

A secondary assumption made by this research is that partial-solution tasks require some minimum amount of processing time to successfully complete. This processing time could include: the time required to successfully complete the current process iteration, the time required to transfer results, or the time required to transfer process control due to an unexpected interruption. Failure to execute for this required amount of time will result in a missed deadline.

The final assumption made by this research is that tasks are independent. This assumption is required by *theore*m *1* of the rate monotonic scheduling. It has also been extended, as a simplifying assumption, to the runtime scheduling phase since it eliminates the need for complex control methods.

3.6 Summary

This chapter has provided some of the methodology used in the development of the pre-runtime and runtime systems for demonstrating the performance benefits of partial-solution tasks. It has summarized the high-level objectives, the test data, the process used for demonstrating the objectives, and proposed four metrics for measuring system performance. Additionally, it has addressed some key design decisions and basic design assumptions. The subsequent chapters in this thesis describe the implementation of the supporting software, test results, and some recommendations for future work.

IV - Implementation of Support Software

This chapter describes the design and implementation of the pre-runtime scheduler and the runtime executive.

The pre-runtime scheduler was developed with two objectives in mind: to create a program for verifying the schedulability of independent periodic tasks; and, to generate fixed-priority schedules for feasible task sets. The pre-runtime scheduler is implemented in Common Lisp and resides on a Sun workstation. The decision to use Common Lisp instead of Ada was based on two factors: First, time constraints limited the amount of time available for developing new software. Second, the Common Lisp scheduler represented a reusable component that required minimal modifications to incorporate into the design of the pre-runtime demonstration.

The runtime executive was developed with one objective in mind - to measure and compare the robustness of systems implemented using partial-solution techniques with the robustness of systems implemented using traditional all-or-nothing techniques. As required by the sponsor, the runtime executive is implemented in Ada. The current version has been developed on a Silicon Graphics Iris workstation using Verdix MP-Ada version 6.21. Selected source code from both systems is contained in Appendix F.

4.1 Pre-Runtime Scheduler

The pre-runtime scheduler was developed to provide static scheduling support for the pre-runtime schedulability and maintainability demonstrations. For a feasible set of tasking requirements, the scheduler produces a fixed-priority schedule. Failure to produce a fixed-priority schedule indicates a non-feasible set of requirements.

The foundation of the scheduler is *theorem 1* of the rate-monotonic scheduling algorithm. According to *theorem 1*, a set of n independent periodic tasks scheduled by the rate-monotonic algorithm will always meet their deadlines, for all task phasings, if the total processor utilization is less than the maximum utilization. In this context, meeting the deadline means that the task will complete prior to its next activation. The theorem is easily extended to multi-processor systems if it is assumed that tasks are independent and CPUs are independent and homogeneous. Such a system is considered schedulable if the set of tasks assigned to each CPU is schedulable.

The scheduling software is divided into four modules: task.lisp, schedule.lisp, print.lisp, and translate.lisp.

4.1.1 task.lisp. This module contains information describing individual task requirements. Each task is defined as a separate entity using *setf*. The *defstruct task* template used for storing task data and an example partial task entry is described in Figure 4.1. The example task can be converted to an all-or-nothing task by replacing the multi-item duration list with a list containing a single duration. Task entities are combined during the scheduling process to create system requirements.

```
                    Task Template
(defstruct task (name nil)
        (duration      nil)
        (period        nil)
        (importance    0)
        (priority      0)
        (bodyin        "./Template/template.adb")
        (bodyout       "./Source/task_body.a")
        (specin        "./Template/template.ads")
        (specout       "./Source/task_spec.a"))
                    Example Task
(setf t1 (make-task
        :name          'task1
        :duration      '(1.5 0.5 0.2)
        :period        '10.0
        :importance    '1
        :bodyout       "./Source/task1_body.a"
        :specout       "./Source/task1_spec.a"))
```

Figure 4.1: Task Template and Sample Task

4.1.2 schedule.lisp. This module is the heart of the scheduling software. It uses a depth-first search strategy to assign tasks to processors and durations to tasks. The depth-first strategy was selected for two reasons: First, the design did not require a strategy capable of generating an optimal solution, but simply one that generated any acceptable solution. Second, the depth-first strategy is easy to implement.

The main entry point into the program is the function *schedule-system*. *Schedule-system* takes as inputs the number of processors allocated to the system and the tasks to be scheduled. The format for calling *schedule-system* is:

```
schedule-system(number-of-processors task1 task2 task3 ... taskN)
```

Schedule-system relies heavily on four other functions to carry out the depth-first search strategy: *schedule-system-aux*, *schedule-tasks*, *schedule-children*, and *schedule-next-task*.

Schedule-system-aux is responsible for initializing the scheduling data structure. Task durations are sorted in descending order to insure higher quality (long duration) processes are considered before lower quality (short duration) processes. Tasks are also sorted by importance to insure higher importance tasks are considered before lower importance tasks. Task priorities are set to zero, and the system processor list is built. Once initialization is complete, control is passed to *schedule-tasks* which tries to schedule the first task in the task list.

Schedule-tasks is responsible for scheduling a single task onto the system. It uses the function *generate-children* to create a list of processes that represent all possible task/duration/processor combinations for the desired task. For example, calling *generate-children* with the task described in Figure 4.1 and two processors would result in six children being placed in the *process-list*: (task1 1.5 1), (task1 1.5 2), (task1 0.5, 1), (task1 0.5, 2), (task1 0.2 1), and (task1 0.2 2). The order of this list determines the order in which processes are evaluated. Once the list of children has been created, control is passed onto *schedule-children*.

Schedule-children is responsible for scheduling a single process on a processor. If *theorem 1* continues to hold when the process is added to the processor, the function will call *schedule-next-task* with a new system list, the old system list, and the next task to be scheduled. If the processor utilization is greater than the maximum utilization, then *schedule-children* will call itself with the first element of the *processes-list* removed. If none of the children can be scheduled, the function will fail and return nil. This function carries out the depth portion of the depth-first search.

Schedule-next-task is responsible for scheduling the next task on the *task-list* onto the new system generated by *schedule-children*. If the next task cannot be scheduled with the new system, the function calls *schedule-children* with the old system and the next process. This function implements the backtracking portion of the depth-first search.

4.1.3 print.lisp. This module is used to print schedules generated by *schedule-system*. It sorts the scheduled tasks by CPU, and period and assigns task priorities by increasing period as described by the rate-monotonic scheduling algorithm. Figure 4.2 provides an example of the output generated by the *print-schedule* routine for the modified GAP requirements.

4.1.4 translate.lisp. This module is used to translate schedules generated by *schedule-system* into Ada source code. The translation is done by reading a template file and making named substitutions to generate the Ada code. Substitutions are limited to one per line and are performed on both specification and body template files. This portion of the scheduler was used during initial development of the runtime executive to minimize the time required for developing Ada tasks.

```
                     Sample Output From Print-Schedule
-> (print-schedule 1 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t12)
----------------------------------------------------------------
Processor #1

     Name             Period Duration Priority Importance Processor
     ---------------  ------ -------- -------- ---------- ---------
     CONTACT_MGNT      0.2    0.05      31         2          1
     TARGET_UPDATE_I   0.5    0.05      32         1          1
     NAV_UPDATE        0.5    0.08      33         1          1
     GRAPHIC_DISPLAY   0.8    0.09      34         3          1
     TARGET_UPDATE_II  1.0    0.05      35         3          1
     KEYSET            2.0    0.01      36         3          1
     STEERING_CMDS     2.0    0.03      37         1          1
     STORES_UPDATE     2.0    0.01      38         1          1
     STATUS_UPDATE     2.0    0.03      39         1          1

     Util     [0.71250000000001]
     Max Util [0.7205376500755]
```

Figure 4.2: Sample Output From Print-Schedule

4.2 Runtime Executive

The runtime executive was developed to support the runtime robustness experiment. It uses generic tasks to simulate the performance of real-time systems developed using all-or-nothing and partial-solutions techniques. The primary purpose of the software is to gather statistical data that measures the effect of randomly generated sporadic tasks on four performance metrics: average solution quality for cyclic tasks, percentage of missed deadlines for cyclic tasks, average solution quality of sporadic tasks, and percentage of missed deadlines for sporadic tasks.

The underlying structure of the executive was developed from examples provided in [Baker, 1989:17]. Baker's original design has been slightly modified to support the additional requirements of sporadic task generation and scheduling. The runtime system is divided into eight components: timer, executive, sporadic scheduler, sporadic generator, cyclic schedule, task interface, display task, and a generic tasks package. Each of these components is encapsulated in an Ada package and described in the following sections. Figure 4.3 illustrates a top-level architecture of the runtime system.

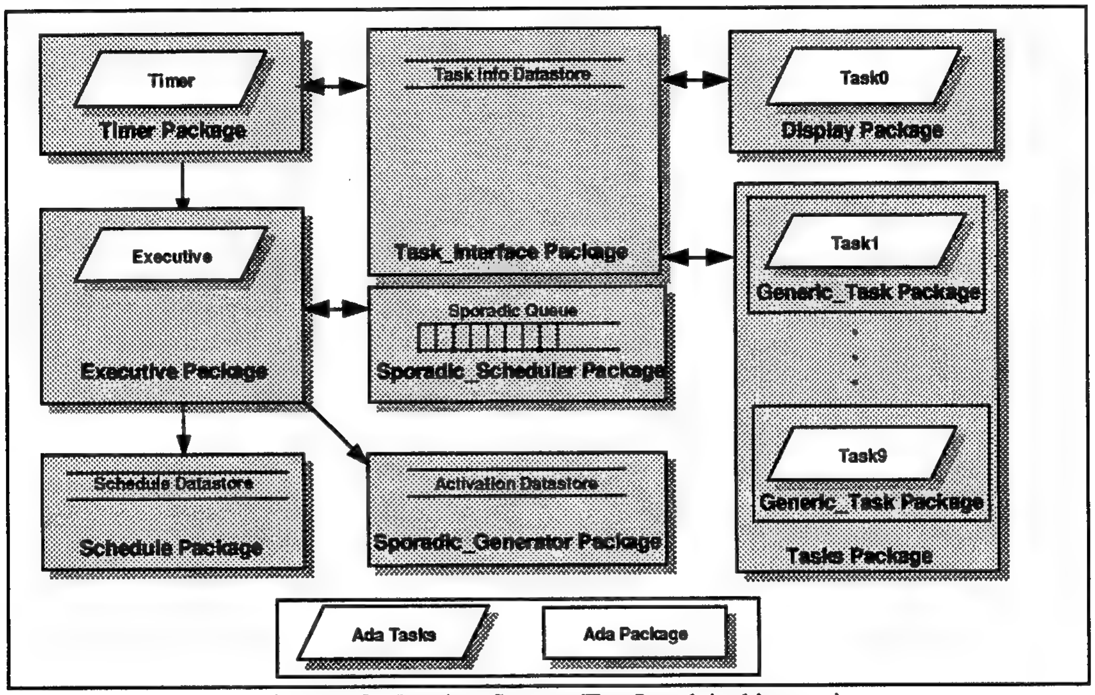


Figure 4.3: Runtime System (Top-Level Architecture)

4.2.1 Package Timer, Executive, and Cyclic Schedule. It is important at this time to make a distinction between Ada tasks and cyclic tasks. The timer task, executive task, display task, and generic tasks are all internally represented using Ada tasking constructs. The simultaneous execution of these tasks on the system processor is conducted using the preemptive fixed-priority scheduling strategy described in the Verdix Ada Language Reference Manual.

The display task and generic tasks are also cyclic tasks. The preemptive execution of cyclic tasks is controlled by the cyclic executive in compliance with the cyclic schedule. Since the current executive has only a single virtual processor, only one cyclic task is allowed to be active at any given time. All other cyclic tasks must be suspended or waiting on an Ada rendezvous with the cyclic executive.

The control structure for the runtime executive revolves around two key components, the cyclic schedule and the timer. The cyclic schedule consists of a fixed number of frames that describe a sequence of events for starting, stopping, suspending, and resuming cyclic tasks. The executive steps through each frame of the cyclic schedule at a rate of one frame every 0.01 second, in response to interrupts generated by the timer. At the end of the schedule the executive resets the frame count to one and repeats the cycle.

During each frame of the cyclic schedule, system processing time is distributed between three Ada tasks: the timer task, the executive task, and the currently running generic cyclic task. At the beginning of the frame, the currently running cyclic task is preempted by the fixed-priority scheduler in response to the activation of the higher-priority timer. The timer resumes the executive task using an Ada rendezvous. Failure of the executive to respond immediately to the rendezvous indicates an executive overrun and is recorded as a missed deadline for the executive. The current technique for handling missed deadlines, except for recording them, is to ignore them. After the rendezvous, the timer resets the alarm, and suspends itself until the next activation.

Following a successful rendezvous between the timer and executive, and after the timer has suspended itself, control is passed to the executive task. The executive performs five actions that determine the next cyclic task to execute:

- It calls the sporadic_generator to update the sporadic activation list.
- It calls the sporadic_scheduler to queue sporadic tasks that have an activation time equal to the current time.
- It calls the sporadic_scheduler to resume any cyclic tasks that were preempted during the previous executive cycle.
- It executes the actions required by the current frame of the cyclic schedule.
- It calls the sporadic_scheduler to determine if a sporadic task should preempt the currently activated cyclic task.

Further details on each of these actions are provided in the next two sections.

After completing the scheduling process, the executive suspends itself and waits for the next rendezvous with the timer. The time remaining, before the next timer activation, is used by the currently activated cyclic or sporadic task to perform work. If no tasks are activated, then the time goes unused.

4.2.2 Package Sporadic_Generator. The sporadic_generator is responsible for generating sporadic task activations at a fixed frequency (number of tasks/sec) and random arrival times for the duration of the simulation. The default value of 10 tasks/second can be modified by passing the desired frequency on the command line using the *-sf* option.

The generator maintains a list of sporadic activations for a period of one second. At the beginning of each second, the activation list is cleared and updated in response to the request from the executive task. The list is filled with the required number of activations using a repeatable pseudo-random number generator. The selection of a repeatable random number generator is important, since it allows the simulation to guarantee the same sequence of sporadic activations from one experiment to the next. The random number generator is used to determine the time index during which the sporadic task should be activated. The activation list is updated by incrementing a counter associated with the requested time index.

4.2.3 Package Sporadic_Scheduler. Activations created by the *sporadic_generator* are consumed by the *sporadic_scheduler*. In response to a request from the executive, the scheduler creates one sporadic task for each activation referenced by the current time index. The creation process consists of setting the task start time, duration, response time, importance, and deadline, and placing the task on the scheduling queue.

Four queuing techniques are available for use by the scheduler: last-in-first-out (LIFO), first-in-first-out (FIFO), deadline, and importance. The only technique used in the partial-solution experiments is the importance-based scheduling. In importance-based scheduling, critical tasks are selected first, essential tasks second, and non-essential tasks last. Importance values for the 11 simulated real-time systems were assigned by the *sporadic_generator*, to the sporadic tasks, *at runtime*, using a repeatable random number generator.

While deadline-driven scheduling has been shown to be optimal in some situations, it does not take into consideration the importance of individual task requirements. Because of this, it is possible that a critical task can be ignored in favor of a non-essential task with a short deadline. In many situations, this may not be appropriate. Importance-based scheduling, while not producing optimal results, places the emphasis on completing tasks that have the greatest impact on the system. Additionally, this strategy allows the scheduler to comply with the Spring multi-level architecture.

Dequeuing is activated by a second request from the executive. In response, the scheduler dequeues all tasks that have executed to completion, passed their deadlines, or require more time than is available before their deadline. The missed deadline count is incremented for those tasks that are dequeued before completion. Following the dequeuing process, the scheduler attempts to schedule the topmost task by preempting the currently scheduled cyclic task. If the processor is idle, or the sporadic task has a higher importance than the cyclic task, then the preemption process succeeds. A successful preemption results in the suspension of the cyclic task and activation of the sporadic task. During the next iteration of the scheduler, the suspended task is resumed and the process repeats.

4.2.4 Package Generic_Task. In the current implementation, sporadic tasks are not implemented as Ada tasks and do not perform any actual work. Instead, they are simulated all-or-nothing tasks that simply act as triggers for suspending the execution of cyclic tasks. On the other hand, cyclic tasks are implemented as Ada tasks and perform work based on the amount of time allocated to them. For all-or-nothing tasks the solution quality of the task is only recorded after the task has successfully completed all its work. For partial-solution tasks, the solution quality is recorded each time the task completes a predetermined section of code.

To minimize the effect of task implementation on experimental results, both the all-or-nothing tasks and partial-solution tasks have been constructed using a single Ada package *generic_task*. This generic package encapsulates an underlying Ada task with the control functions necessary for initializing, starting, stopping, resuming, suspending, stop-working (stopping the task prior to completion), and terminating the cyclic task in response to commands from the executive. Each cyclic task is simply an instantiation of the *generic_task*, with the *task_id* defined at compilation time and the *task_type* defined at runtime.

```
---------------------------------------------------------------
-- Reference to conservative can be modified to produce alternate
-- performance curves (linear, optimistic)
---------------------------------------------------------------
task body doit is
   temp      : integer := 1;
   iteration : integer := 0;
begin
   task_id := v_i_tasks.get_current_task;
   accept initialize_task;

   loop
       accept start_next_cycle do
          completed    := FALSE;
          stopwork     := FALSE;
          reset        := TRUE;
          iteration    := 0;
          loops(id)    := 0;
          quality(id)  := 0.0;
      end start_next_cycle;

       while stopwork = FALSE loop
          if reset then
             reset := FALSE;
             iteration    := 0;
             loops(id)    := 0;
             quality(id)  := 0.0;
         end if;

          for count in 1..lcount  loop
             if stopwork or reset then
                exit;
            end if;
             temp := (temp + 1) mod 101;
             loops(id) := loops(id) + 1;
         end loop;

          iteration    := iteration + 1;
          if ttype(id) = ITERATIVE_TASK then
             quality(id) :=  conservative(iteration);
         end if;
          exit when iteration = 20;
       end loop;

       if (ttype(id) = ITERATIVE_TASK) then
          if (iteration /= 0) then
             quality(id) :=  conservative(iteration);
         end if;
      else
          if (iteration = 20) then
             quality(id) := 1.0;
         end if;
      end if;

       completed := TRUE;

   end loop;
 end doit;
```

Figure 4.4: Underlying Ada Task

The implementation of the underlying Ada task is based on the Root-By-Newton method for generating square roots. Root-By-Newton uses an iterative process to increase the quality of a guess during each consecutive iteration. The process terminates when the change in the guess is within a specified tolerance. Since the research example only tries to find a single real, positive square root, to a real positive number, convergence is not considered a problem.

The algorithm is representative of the general class of partial-solution tasks, because it can produce an approximation (partial-solution) even if it is terminated prior to meeting the tolerance requirement. The implemented version has been modified from its original form to support additional requirements. These requirements are a result of information gained during the runtime test and are further addressed in Chapter V.

In its current form, the software does not actually perform the square-root function. Instead, it uses a performance curve and a *worker loop* to simulate the processing cost and solution quality of the function (see Figure 4.4). This method increases the flexibility of the software, by allowing it to model several functions without having to develop additional, and sometimes complex, Ada tasks.

The distinction between all-or-nothing tasks and partial-solution tasks is embodied in the method used for recording the task solution quality. At the start of each task, the solution quality is initialized to 0.0. For partial-solution tasks, the solution quality is recorded at the end of each iteration. For all-or-nothing tasks, the quality is modified to 1.0 only if the task runs to completion within the allocated time.

While the solution quality of the task is fixed, the processing cost can be varied by scaling the duration of the *worker loop*. For instance, a task requiring a duration of 0.20 seconds may require the *lcount* be set to 70,000 while one requiring 0.02 seconds only requires the loop to execute for 7,000 (the *lcount* variable can be set using the command line option *-l*). This capability also functions as a calibration technique when porting the simulation between several machines with different processing capabilities.

4.2.5 Package Task_Interface.

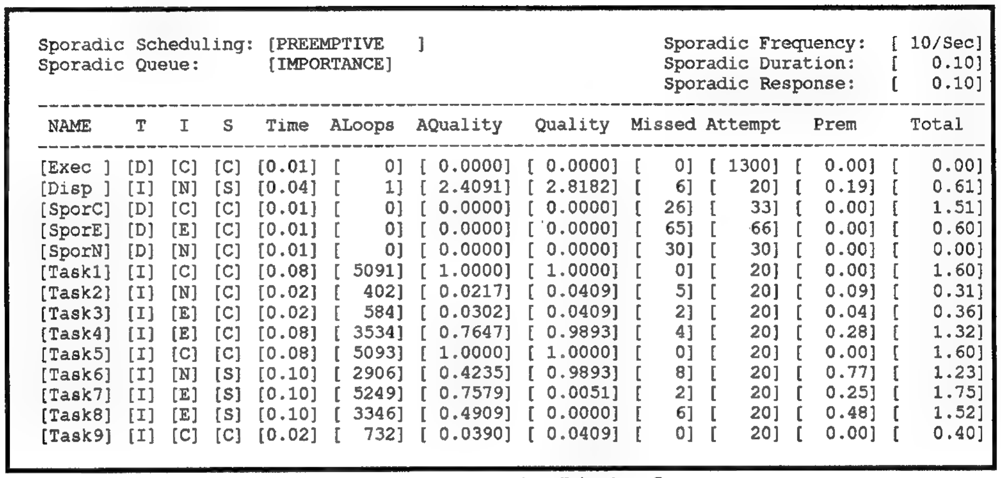
Communications between the cyclic tasks and the executive are provided via the *task_interface* package. The interface was developed to minimize the number of packages requiring references to the generic tasks. It uses a single function, *update_task* to pass control commands from the executive to all system tasks: executive, timer, display, cyclic and sporadic.

As well as handling task communications, the interface package also contains the central data repository for system data and task statistics. For each task it maintains: task id, task type, task name, task duration, task importance, task status, task activation flag, number of missed deadlines, number of attempted deadlines, current solution quality, total solution quality, average solution quality, current number of worker loops, total number of worker loops, average number of worker loops, and total time used. For sporadic tasks it maintains: sporadic duration, sporadic response time, and sporadic frequency. For the scheduler it maintains: current scheduling mode, running task id, interrupted task id and a nonpreemptive scheduling flag. Finally, for the system it maintains the test duration and interactive mode flag.

4.2.6 Package Display. Much of the information maintained in the interface package can be displayed using facilities provided by package *display*. *Display* contains a cyclic task modified to provide both interactive and non-interactive data presentation.

The interactive display uses *curses*-based functions to present system and task information during runtime. Curses is UNIX code library that provides device independent routines for terminal IO. Interestingly enough, it was necessary to implement the interactive display using a partial-solution task. In its original version, the display I/O was implemented using the TEXT_IO functions provided in package *standard*. These functions were unable to refresh the entire screen in the allocated time. Instead of increasing the amount of time available to the display tasks, it was modified to perform partial updates in an interactive manner (much like the iterative example provided in Chapter III). The resulting code achieved an update rate (or solution quality) of approximately 0.3 screens per execution. When the code was modified to use the curses functions, the update rate jumped to approximately eight to ten screens per execution. Example output from the interactive display is provided in Figure 4.5.

The non-interactive version of the display uses TEXT_IO functions to output simulation statistics to the UNIX standard output pipe (stdout). By using stdout, it is possible to redirect the statistical data directly into a file, thereby eliminating the need for human interaction. This capability reduces the amount of time required for gathering and recording simulation data. Output from the non-interactive display is only generated for cyclic tasks and is limited to the task solution quality and number of missed deadlines.



```
Sporadic Scheduling: [PREEMPTIVE   ]                       Sporadic Frequency:  [ 10/Sec]
Sporadic Queue:      [IMPORTANCE]                          Sporadic Duration:   [   0.10]
                                                           Sporadic Response:   [   0.10]
------------------------------------------------------------------------------------------
 NAME    T   I   S   Time  ALoops  AQuality   Quality  Missed Attempt   Prem     Total
------------------------------------------------------------------------------------------
[Exec ] [D] [C] [C] [0.01] [     0] [ 0.0000] [ 0.0000] [   0] [ 1300] [  0.00] [   0.00]
[Disp ] [I] [N] [S] [0.04] [     1] [ 2.4091] [ 2.8182] [   6] [   20] [  0.19] [   0.61]
[SporC] [D] [C] [C] [0.01] [     0] [ 0.0000] [ 0.0000] [  26] [   33] [  0.00] [   1.51]
[SporE] [D] [E] [C] [0.01] [     0] [ 0.0000] [ 0.0000] [  65] [   66] [  0.00] [   0.60]
[SporN] [D] [N] [C] [0.01] [     0] [ 0.0000] [ 0.0000] [  30] [   30] [  0.00] [   0.00]
[Task1] [I] [C] [C] [0.08] [  5091] [ 1.0000] [ 1.0000] [   0] [   20] [  0.00] [   1.60]
[Task2] [I] [N] [C] [0.02] [   402] [ 0.0217] [ 0.0409] [   5] [   20] [  0.09] [   0.31]
[Task3] [I] [E] [C] [0.02] [   584] [ 0.0302] [ 0.0409] [   2] [   20] [  0.04] [   0.36]
[Task4] [I] [E] [C] [0.08] [  3534] [ 0.7647] [ 0.9893] [   4] [   20] [  0.28] [   1.32]
[Task5] [I] [C] [C] [0.08] [  5093] [ 1.0000] [ 1.0000] [   0] [   20] [  0.00] [   1.60]
[Task6] [I] [N] [S] [0.10] [  2906] [ 0.4235] [ 0.9893] [   8] [   20] [  0.77] [   1.23]
[Task7] [I] [E] [S] [0.10] [  5249] [ 0.7579] [ 0.0051] [   2] [   20] [  0.25] [   1.75]
[Task8] [I] [E] [S] [0.10] [  3346] [ 0.4909] [ 0.0000] [   6] [   20] [  0.48] [   1.52]
[Task9] [I] [C] [C] [0.02] [   732] [ 0.0390] [ 0.0409] [   0] [   20] [  0.00] [   0.40]
```

Figure 4.5: Interactive Display Output

4.3 Summary

This chapter described the software architectures used to conduct the pre-runtime demonstrations and the runtime experiments. In Chapter V, results from the experiments are analyzed to determine if actual results reflect expected results. Part of this analysis includes a review of the software implementation and its impact on experimental results.

V-Test Plan and Results

Chapter III described the methodology used by this research to investigate the potential benefits of partial-solution tasks. This chapter develops a test plan based on that methodology, presents results from the execution of the test plan, and analyzes the results to determine if the partial-solution tasks performed as expected. In response to unexpected results, some extensions were made to the original test plan. To preserve the flow of reasoning, these extensions are presented chronologically in the results and analysis section. Results gathered during the first (scheduled) phase of the runtime experiment are referred to as *initial results*, while results gathered during the second (unscheduled) phase are referred to as *extended results*.

5.1 Test Plan

The performance test plan is divided into two sections: the pre-runtime schedulability and maintainability demonstration, and the runtime robustness test. Each section identifies the expected results and the process used to generate actual results.

5.1.1 Pre-Runtime Schedulability and Maintainability Demonstration. Demonstrating the increased schedulability and maintainability of partial-solution tasks was conducted in three phases: In the first phase, the original GAP timing requirements (detailed in Table 3.1) were scheduled using the rate-monotonic scheduler. Results from the scheduler were expected to reflect that the timing requirements are non-schedulable using *theorem 1* of the rate-monotonic scheduling algorithm. In the second phase, the GAP Graphics-Display task was modified to reflect implementation as a partial-solution task. This task was selected for its high utilization of 0.113. Results from scheduling the new requirements using the rate-monotonic scheduler was expected to demonstrate the increased schedulability of partial-solution tasks by generating a feasible schedule where none existed using all-or-nothing tasks. And in the final phase, additional tasks were added to the modified GAP to simulate increased requirements due to maintenance (i.e., upgrading software). Results from scheduling each additional task using the rate-monotonic scheduler was expected to demonstrate the increased maintainability of partial-solution tasks by generating feasible schedules that degrade the performance of the Graphics-Display task to satisfy the requirements of the new task.

5.1.2 Runtime Robustness Test. Testing the increased robustness of partial-solution tasks was accomplished by experimentally measuring the effect of random sporadic task arrivals on eleven simulated real-time systems. Ten of these systems consisted of randomly generated timing requirements while the eleventh contained the modified GAP requirements presented in Table 3.2. Detailed timing information for the randomly generated systems can be found in Appendix A.

The performance of the eleven systems was measured using four metrics: average solution quality for cyclic tasks (metric 1), number of missed deadlines for cyclic tasks (metric 2), average solution quality for sporadic tasks (metric 3), and number of missed deadlines for sporadic tasks (metric 4). For partial-solution cyclic tasks, the solution quality was calculated using the performance curves shown in Figure 5.1. These three curves were selected because they are representative of three types of processes. The

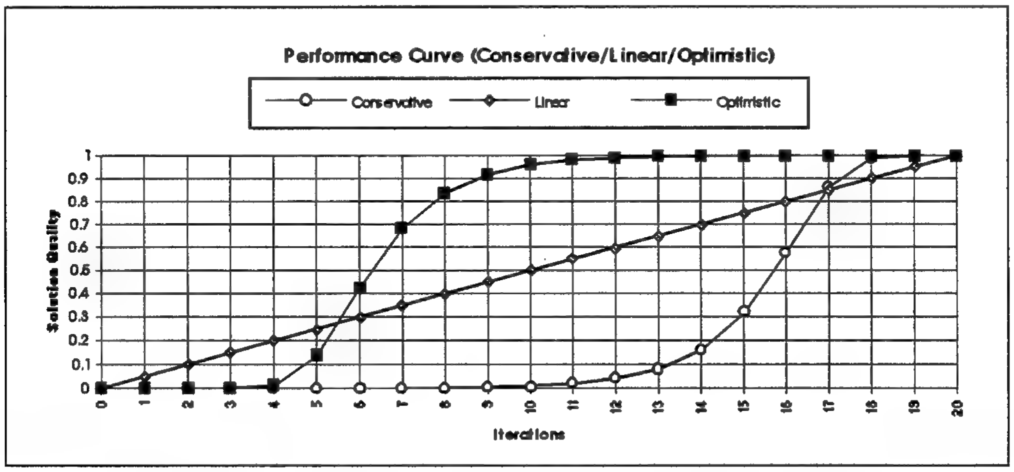


Figure 5.1: Conservative/Linear/Optimistic Performance Curves

Conservative curve represents processes that achieve only a small change in solution quality early in their execution but a large degree of change during the later portion of their work. The Optimistic (reverse Conservative) curve describes the opposite situation, where the process achieves most of its accuracy early in process execution and makes only slight improvements as additional time is used. The Linear curve represents a process whose solution quality changes at the same rate throughout the entire execution process.

Since the sporadic tasks are simulating all-or-nothing tasks, they can only achieve solution qualities of zero or one. Based on this, their average quality can be calculated by dividing the number of missed deadlines by the number of activations.

The runtime characteristics of the sporadic tasks are described using four attributes: duration, response time, frequency and importance. Since it is not feasible to test all possible combinations of these attributes, a cross-section of the population was selected using three tests. In the first test, frequency was varied, while duration and response time were kept constant. In the second test, duration was varied, while frequency and response time were kept constant. And, in the final test, response time was varied, while frequency and duration were kept constant. During all tests the importance of the sporadic tasks was randomly generated. In total, nine tests were conducted on the eleven simulated systems:

- Conservative Performance - Fixed Frequency.Test
- Conservative Performance - Fixed Duration Test
- Conservative Performance - Fixed Response Test
- Linear Performance - Fixed Frequency Test
- Linear Performance - Fixed Duration Test
- Linear Performance - Fixed Response Test
- Optimistic Performance - Fixed Frequency Test
- Optimistic Performance - Fixed Duration Test
- Optimistic Performance - Fixed Response Test

Each of the performance tests was executed twice: the first time using simulated all-or-nothing tasks and the second time using partial-solution tasks. Results from these two runs were compared to determine if the partial-solution tasks outperform the all-or-nothing tasks. It was expected that the partial-solution tasks would decrease the number of missed deadlines and increase the average solution quality in each of the eleven systems.

5.2 Chronological Test Results

The following section summarize results for both the pre-runtime demonstration and the runtime experiment. Full results are available in the appendices. Results from the pre-runtime demonstration are available in Appendix B, while results from the initial runtime experiment are shown in Appendix C, extended runtime results in Appendix D, and group runtime results in Appendix E.

5.2.1 Infeasible GAP Schedule. During phase one of the schedulability and maintainability demonstration, the GAP schedule presented in Table 3.1 was shown to be infeasible using *theorem 1* of the rate-montonic scheduling algorithm. The schedule became infeasible when the eighth task (Status-Update-I) was added, causing the total utilization to exceed the eight-task upper bound of 0.7241.

5.2.2 Feasible GAP Schedule. During the second phase of the schedulability and maintainability demonstration, the GAP *Graphics-Display* process was modified to simulate implementation as a partial-solution task. The modification consisted of replacing the single-field duration of (0.009) with a multi-field duration of (0.001 0.002 0.003 0.004 0.005 0.006 0.007 0.008 0.009). This new duration was designed to simulate implementation of an interruptible iterative task that requires a closure/clean-up time of 0.001 seconds.

As expected, the rate-monotonic scheduler was able to generate a feasible schedule using the new partial-solution task. The resulting fixed-priority schedule, which is shown in Figure 5.2, was possible because the scheduler was able to degrade the requirements of the *Graphics-Display* task to 0.005 seconds. The new solution quality of the task (before runtime preemption) can be determine by cross referencing the task duration with the task performance curve.

```
-> (print-schedule 1 t1 t2 t3 t4m t5 t6 t7 t8 t9 t10 t11 t12 t13)
-----------------------------------------------------------------
Processor #1

     Name               Period  Duration  Importance  Processor
     ----------------   ------  --------  ----------  ---------
     CONTACT_MGNT       0.025   0.005     2           1
     TRACKING_FILTER    0.025   0.002     1           1
     TARGET_UPDATE_I    0.05    0.005     1           1
     NAV_UPDATE         0.059   0.008     1           1
     HOOK_UPDATE        0.065   0.002     1           1
     GRAPHIC_DISPLAY    0.08    0.005     3           1
     TARGET_UPDATE_II   0.1     0.005     3           1
     KEYSET             0.2     0.001     3           1
     STEERING_CMDS      0.2     0.003     1           1
     STORES_UPDATE      0.2     0.001     1           1
     STATUS_UPDATE-I    0.2     0.003     1           1
     NAV_STATUS         1.0     0.001     1           1
     STATUS_UPDATE_II   1.0     0.001     1           1

     Util      [0.7008624511082139]
     Max Util [0.7119589942614066]
```

Figure 5.2: Feasible GAP Schedule

5.2.3 Increased Requirement GAP Schedule.

During the final phase of the schedulability and maintainability demonstration, the modified GAP was supplemented with four additional tasking requirements based on a subsystem from the Air Force's F-22 project. These new requirements, which are shown in Table 5.1, represent additional task loads that might be added to an avionics platform during software maintenance. The total utilization presented in the table include the utilization from the original GAP tasks.

Table 5.1: GAP Maintenance Requirements

Task Name	Period	Duration	Utilization	Total Utilization
Collision Monitor	0.200	0.0003	0.0015	0.7535
Pilot Manager	1.000	0.0081	0.0081	0.7616
Tactical Manager	1.000	0.0269	0.0269	0.7885
EOB Manager	1.000	0.1331	0.1331	0.9216

The impact of these additional tasks and the adaptive response of the rate-monotonic scheduler are summarized in Table 5.2. The results show that the scheduler can adapt the performance of the Graphic-Display to accommodate the first three maintenance requirements. However, the scheduler was

Table 5.2: Results from GAP Maintenance Requirements

```
Name                 Period  Duration  Priority  Importance  Processor

GRAPHIC_DISPLAY      0.08    0.005     36        3           1
COLLISION_MONITOR    0.2     0.0003    42        1           1
Util      [0.7023624511082137]
-------------------------------------------------------------------
GRAPHIC_DISPLAY      0.08    0.004     36        3           1
COLLISION_MONITOR    0.2     0.0003    42        1           1
PILOT_MANAGER        1.0     0.0081    45        1           1
Util      [0.6979624511082138]
-------------------------------------------------------------------
GRAPHIC_DISPLAY      0.08    0.002     36        3           1
COLLISION_MONITOR    0.2     0.0003    42        1           1
PILOT_MANAGER        1.0     0.0081    45        1           1
TACTICAL_MANAGER     1.0     0.0269    46        1           1
Util      [0.6998624511082138]
-------------------------------------------------------------------
EOB_MANAGER          1.0     0.1331    32        1           1
GRAPHIC_DISPLAY      0.08    0.009     35        3           2
COLLISION_MONITOR    0.2     0.0003    41        1           2
PILOT_MANAGER        1.0     0.0081    44        1           2
TACTICAL_MANAGER     1.0     0.0269    45        1           2
Util      [0.2686932203389830] (Processor 1)
Util      [0.6517692307692307] (Processor 2)
```

not able to degrade the system enough to schedule all four tasks on a single processor. Adding an additional processor to the system alleviated the problem and allowed the schedule to automatically maximize the performance of the Graphic-Display in response to the new resources. Had the Graphic-

Display been implemented as an all-or-nothing task, it would have been impossible to support any of the intermediate requirements without requiring the additional processor.

5.2.4 **Initial Runtime Test Results.** A summary of the runtime test results is presented in Table 5.3. This table classifies results by task performance curve, test type, metric, and task requirements (GAP (G), random (R), or both (B)). Results are shown as positive (+), negative (-), and unchanged (~). A positive result indicates an improved solution quality or decrease in the number of missed deadlines. A negative result indicates a degraded solution quality or increase in the number of missed deadlines. The last two rows of this table sum the total number of tests, by requirement, that produced positive, negative, and unchaged results.

Table 5.3: Runtime Results

		Cyclic-Quality (metric 1)			Cyclic-Deadlines (metric 2)			Sporadic-Quality (metric 3)			Sporadic-Deadlines (metric 4)		
Curve	Test	+	~	-	+	~	-	+	~	-	+	~	-
Conservative	Duration			B	B				B			B	
	Frequency			B	B				B			B	
	Response			B	B				B			B	
Linear	Duration			B	B				B			B	
	Frequency		B		B				B			B	
	Response		B		B				B			B	
Optimistic	Duration	R	G		B				B			B	
	Frequency	B			B				B			B	
	Response	B			B				B			B	
	Random Total	3	2	4	9	0	0	0	9	0	0	9	0
	GAP Total	2	3	4	9	0	0	0	9	0	0	9	0

Information shown in Table 5.3 reflects both expected and unexpected results. The expected result is the decrease in cyclic-deadlines (metric 2). In all nine cases, this metric measured a decrease in the percentage of missed deadlines being experienced by the cyclic tasks. Figure 5.3 provides a graphical description of the performance increases during the Conservative-Frequency test. While these graphs only show results from a single test, they do illustrate the general trend experienced during all nine tests. All results not presented in this section, along with a key to the results, is available in Appendix C.

The *sporadic frequency vs percent missed-deadlines* graphs describes the percentage of activated cyclic tasks that missed their deadlines. Both of these graphs show that systems using partial-solution tasks experiencing fewer missed deadlines than the same systems using all-or-nothing tasks. The

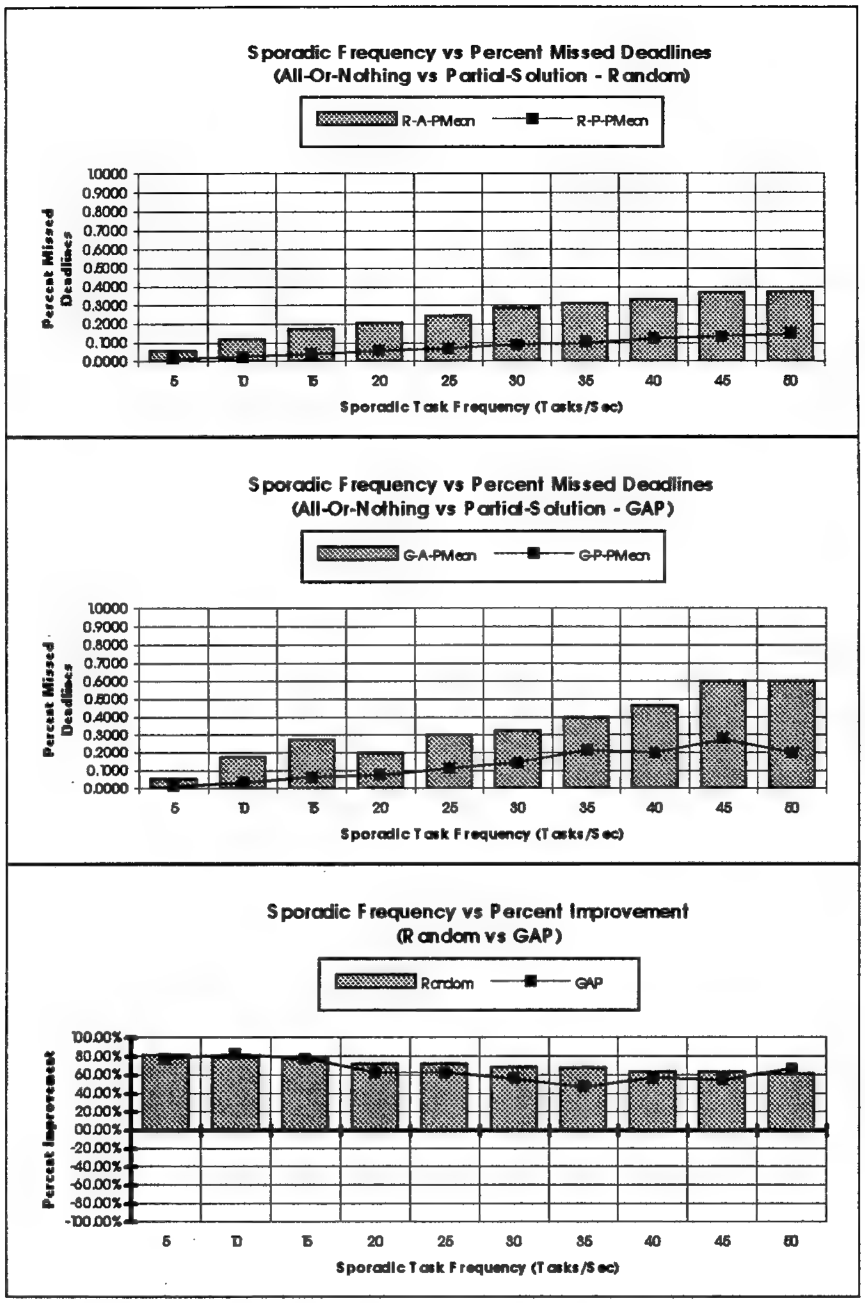


Figure 5.3: Conservative-Frequency Metric 2

sporadic frequency vs percent improvement graph quantifies the reduction in missed deadlines achieved by using partial-solution tasks. For the Conservative-Frequency test, the reduction was 45% to 80% for the GAP requirements and 60% to 80% for the randomly generated requirements.

Visual analysis of the group data presented in Appendix E (Figures E.1 through E.6) indicate no correlation between the decrease in cyclic-deadlines (metric 2) and the shape of task performance curves. For each curve (Conservative, Linear, and Optimistic) the measured performance was nearly identical.

The slight deviations that were observed maybe contributed to a lack of tight process control under the UNIX operating system. Performance results did vary slightly between duration, frequency and response tests. However, on average, the partial-solution tasks were able to decrease the number of missed deadlines by 70% for the randomly generated requirements and 60% for the GAP requirements.

The first unexpected result, shown in Table 5.3, is the decrease in cylic-quality (metric 1). In the three Conservative tests, the average solution quality of the system using partial-solution tasks ranged from 20% to 60% worse than the same system using all-or-nothing tasks. In the three Linear tests, partial-solution tasks resulted in neither an increase nor decrease in solution quality. Only in the Optimistic tests do the results show increased performance from implementation as partial-solution tasks. The range of these results can be clearly seen in Figures 5.4 and 5.5.

Figure 5.4 shows the effect of partial-solution tasks during the Conservative-Frequency test. In this test, the *sporadic frequency vs solution quality* graphs clearly show the partial-solution tasks producing lower quality solution than the all-or-nothing tasks. The quantified results in the *sporadic frequency vs percent improvement* graph show that the randomly generated requirements achieved solution qualities 20% to 25% worse than the all-or-nothing tasks. The GAP requirements did even worse, with solution qualities as much as 60% lower than the all-or-nothing tasks. Furthermore, the slope of the GAP curve indicates that the degradation in solution quality becomes increasingly worse as requirements increase. This characteristic suggests that the system is less robust than its all-or-nothing counterpart.

Figure 5.5 shows the effect of partial-solution tasks during the Optimistic-Frequency test. In this test, the *sporadic frequency vs solution quality* graphs show the partial-solution tasks producing higher quality solutions than the all-or-nothing tasks. Furthermore, the *sporadic frequency vs percent improvement* graph shows a general trend of increased robustness for both the GAP and randomly generated timing requirement.

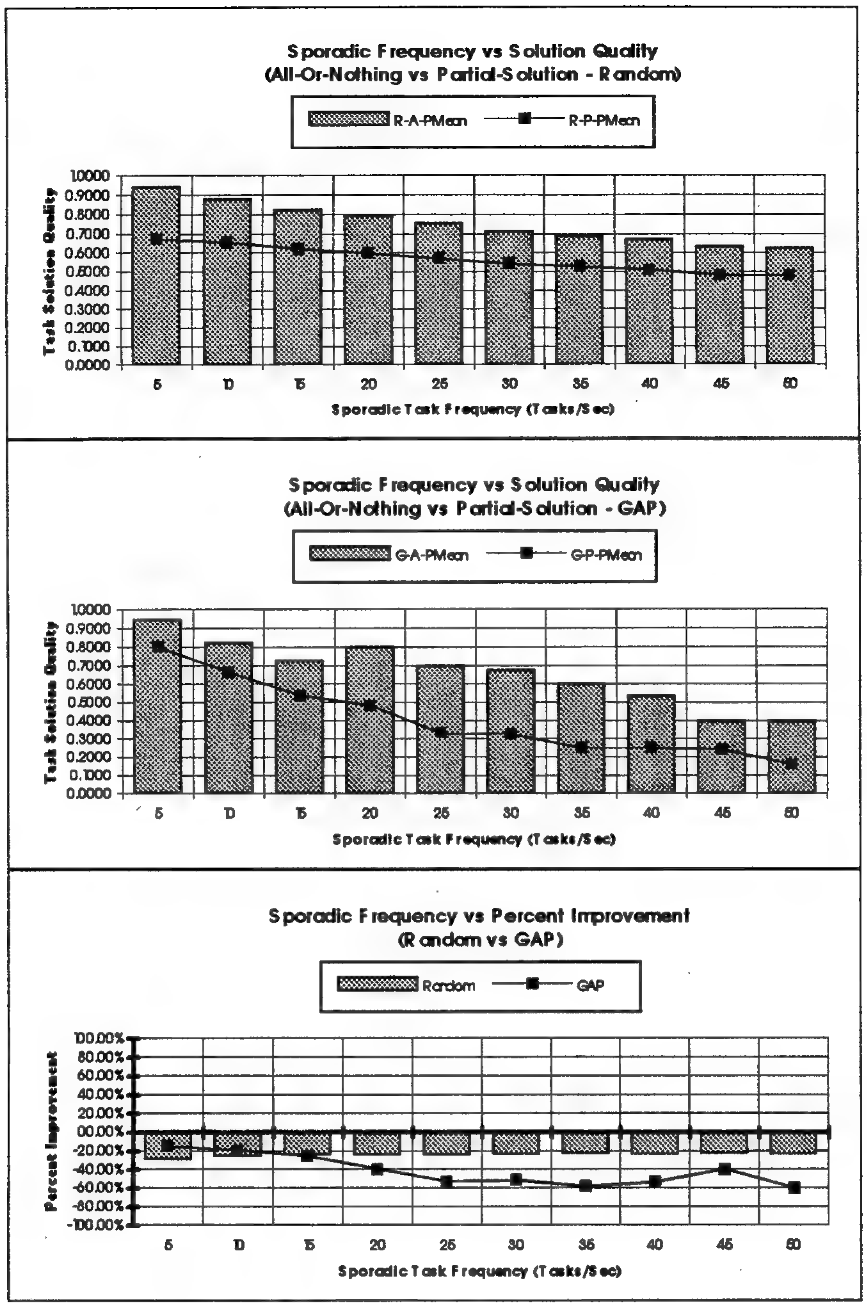


Figure 5.4: Conservative-Frequency Metric 1

The disparity between these results does suggest an important relationship between the effect of partial-solution tasks and task performance curves. Better performance appears to occur in systems that achieve large increases in solution quality during early task execution. This relationship is reflected in the increasing performance from the Conservative curve to the Linear curve, and the Linear curve to the Optimistic curve. Since performance curves are task specific, the data also suggest that best results can be achieved using domain-specific information.

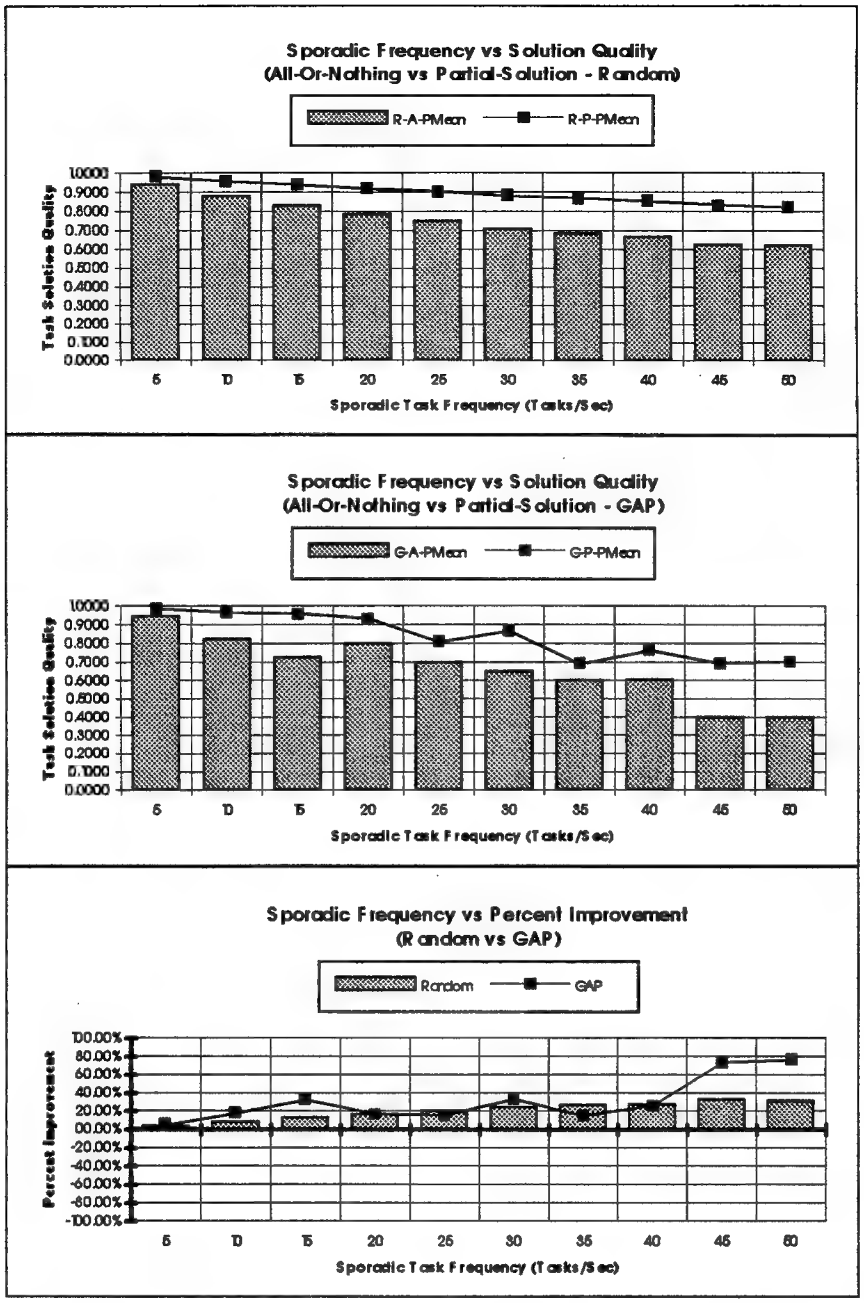


Figure 5.5: Optimistic-Frequency Metric 1

Another unexpected result was the information reflected by sporadic-quality (metric 3) and sporadic-deadlines (metric 4). In all nine test cases, these two metrics showed no change between the performance of the partial-solution tasks and the performance of the all-or-nothing tasks. After some analysis, it was determined that these results are due to implementation of the sporadic scheduler. The scheduler uses a simple importance-based queue to decide when to preempt executing tasks. Since task arrivals have not changed, the scheduler is making identical decisions about which cyclic tasks to preempt and which sporadic tasks to schedule. Addition of an intelligent scheduler should changes this result.

Overall, results from scheduled phase of the runtime test reflect varying performance benefits. Cyclic-quality (metric 1) results suggest that partial-solution tasks **may not** be appropriate for systems trying to decrease missed deadlines and increase average solution quality, depending on how fast solution quality falls off. However, decreased cyclic-deadlines (metric 2) suggests that partial-solution tasks can be extremely beneficial in systems that place more importance on meeting deadlines even if it results in a lower quality solution.

5.2.5 Analysis of Cyclic-Quality Results. In response to the unexpected results, an effort was made to analyze the runtime executive, to ascertain if its design contributed to the decreased solution quality exhibited during the Linear and Conservative tests. From the analysis, it was determined that the degraded performance was caused by three interacting factors: the assumption that partial-solution tasks require a closure/clean-up time, the method used for implementing the generic_task, and the time resolution of the executive.

The generic_task was developed to simulate the execution of both all-or-nothing and partial-solution tasks. To accomplish this, it provides several control functions that allow the runtime executive to initialize, start, stop suspend, resume, terminate, and abort cyclic tasks. The only function not shared between all-or-nothing tasks and partial-solution tasks is the stop-work function, which is used to stop a partial-solution task prior to normal completion.

Stop-work is required to support the assumption that partial-solution tasks require a closure/clean-up time. It is based on the expectation that partial-solution tasks will routinely be required to stop prior to completion. This differs from all-or-nothing tasks, where successful completion is the norm and failure rarely happens. When a failure does happen, the runtime executive must determine when and how to reset the failed tasks, a process that uses valuable resources that may already be allocated to other tasks. Handling partial-solution tasks in the same manner as all-or-nothing tasks could create considerable overhead for the runtime executive and cause degraded performance. Accounting for the closure time in advance limits the stress on the executive and subsequent impact on system performance.

Ada83 provides two methods to cause a task to abandon its normal execution path: *abort* and *exception*. However, both of these methods have limitations that make them inappropriate for use in the runtime software [Baker, 1989:19]. First, no method exists for passing an exception from one task to another. A method did exist in the 1980 version of the Ada Language Reference Manual, but it was removed in 1983. Second, using the *abort* command results in a terminated process that cannot be restarted without using extremely slow task creation.

Overcoming the deficiencies of Ada83 can be accomplished using one of two techniques. The first technique allows the executive task to create an exception in the cyclic task by setting some value in a manner that would cause an exception, such as setting the dividend of the division operation to zero to cause a floating point error. To use this technique, the cyclic task must setup checkpoints throughout its code to generate the exception. This process is both inefficient and a poor software design. The second technique allows the executive task to set a flag in the cyclic task. This technique still requires the cyclic task to setup checkpoints, but allows the task to handle the early stopping without costly Ada exceptions and out of place division operations. The code segment in Figure 5.6 illustrates the latter options used in implementing the generic-task.

```
task body doit is
   --local variables
begin
   accept initialize_task;

   loop
      accept start_next_cycle do
         -- do initialization
      end start_next_cycle;

      while stopwork = FALSE loop
         -- do work exit when done
      end loop;
      -- do clean-up
  end loop;
end doit;
```

Figure 5.6: Stop-Work Code Segment

Since the runtime executive never really knows where in the execution loop the cyclic tasks are, it must assume the worst case, where the task has just begun executing when the stop-work flag is set.

Therefore, the executive must set the stop-work flag a minimum of one loop duration prior to the desired stop time. However, the executive can only issue commands at fixed intervals (usually imposed by hardware and software constraints). Because of this, the stop-work command can only be issued at times that are a multiple of the executive resolution. The difference between the time resolution of the executive and the loop duration can result in lost performance.

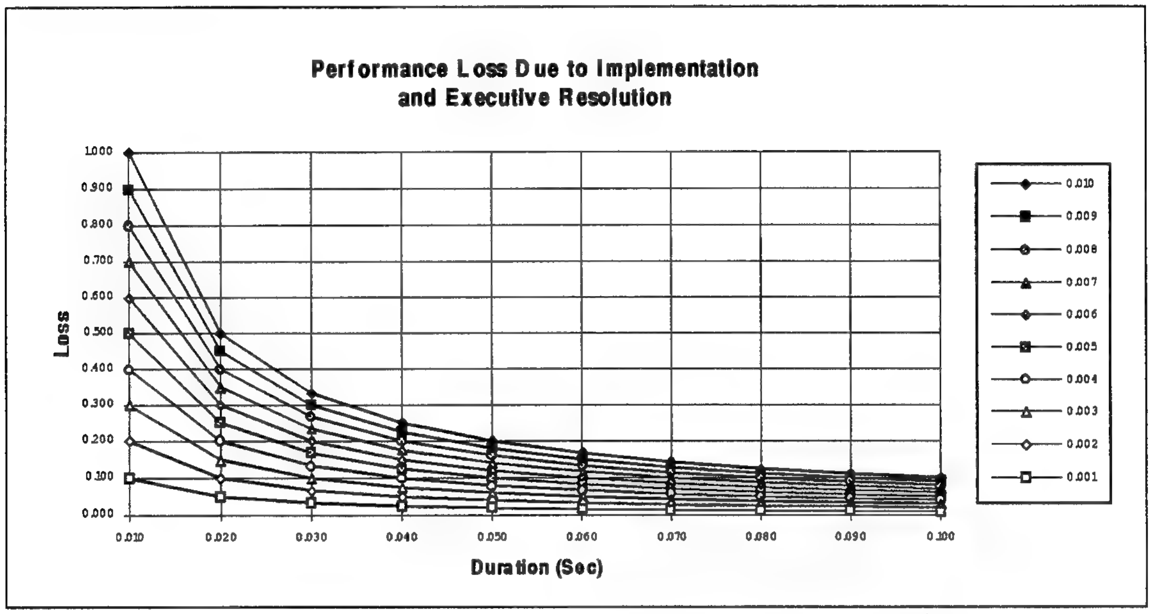


Figure 5.7: Partial-Solution Performance Loss Due to Implementation

Figure 5.7 graphs the lost performance for several resolutions and task durations. This graph measures the amount of time that is lost, not a loss in solution quality. Determining the lost solution quality requires locating the original time minus the lost time on to the task performance curve. With an executive resolution of 0.01 second and task durations less than 0.05 second, partial-solution tasks lost more than 80% of their processing time. Translated to the Linear curve, this restricts the partial-solution processes to achieving a solution quality no higher than 0.20. The Conservative curve is even worse, with a maximum quality of 0.01. This reduction in solution quality is the cause of the degraded performance measured by metric 1.

5.2.6 Extended RuntimeTest Results. In response to the original analysis results, the test plan was extended to include nine additional tests. The new tests were designed to simulate the impact of a small amount of domain knowledge on the performance of the partial-solution tasks. The new tests were identical in form to the original tests, but limited implementation of partial-solution tasks to those cyclic tasks with durations greater than 0.05 second. Selecting 0.05 second was based on the time resolution of the executive and the performance loss graph shown in Figure 5.7. This time represents the knee of the curve where loss due to implementation starts to flatten out.

Results from the nine extended tests are shown in Table 5.4 and Appendix D. These results show improvements in metric 1, over the initial test, for both the Conservative and Linear curves.

Table 5.4: Extended Test Results

		Cyclic-Quality (metric 1)			Cyclic-Deadlines (metric 2)			Sporadic-Quality (metric 3)			Sporadic-Deadlines (metric 4)		
Curve	Test	+	~	-	+	~	-	+	~	-	+	~	-
Conservative	Duration		B		B				B			B	
	Frequency		B		B				B			B	
	Response		B		B				B			B	
Linear	Duration	B			B				B			B	
	Frequency	B			B				B			B	
	Response	B			B				B			B	
Optimistic	Duration	B			B				B			B	
	Frequency	B			B				B			B	
	Response	B			B				B			B	
	Random Total	6	3	0	9	0	0	0	9	0	0	9	0
	GAP Total	6	3	0	9	0	0	0	9	0	0	9	0

Test results for the extended Conservative tests show a positive increase in solution quality for both the randomly generated and GAP requirements. However, even with the application of simulated domain knowledge, the partial-solution tasks are still unable to perform better than the all-or-nothing tasks. Results from the Linear tests show marginally better performance for the randomly generated requirements. These 10 systems moved from the unchanged category into the positive category by achieving a 20% increase in solution quality over the equivalent all-or-nothing systems.

The most significant improvements measured by metric 1 occur in the Linear-Frequency and Optimistic-Frequency extended tests. In these two tests, the solution quality of the GAP requirements was increased on average by 50%. In some situations, these two tests achieved improvements as high as 90% to 120% better than their all-or-nothing counterparts. In addition, both tests, which are shown in Figures 5.8 and 5.9, suggest increased robustness since the percentage of improvement tends to increase as the requirements increase.

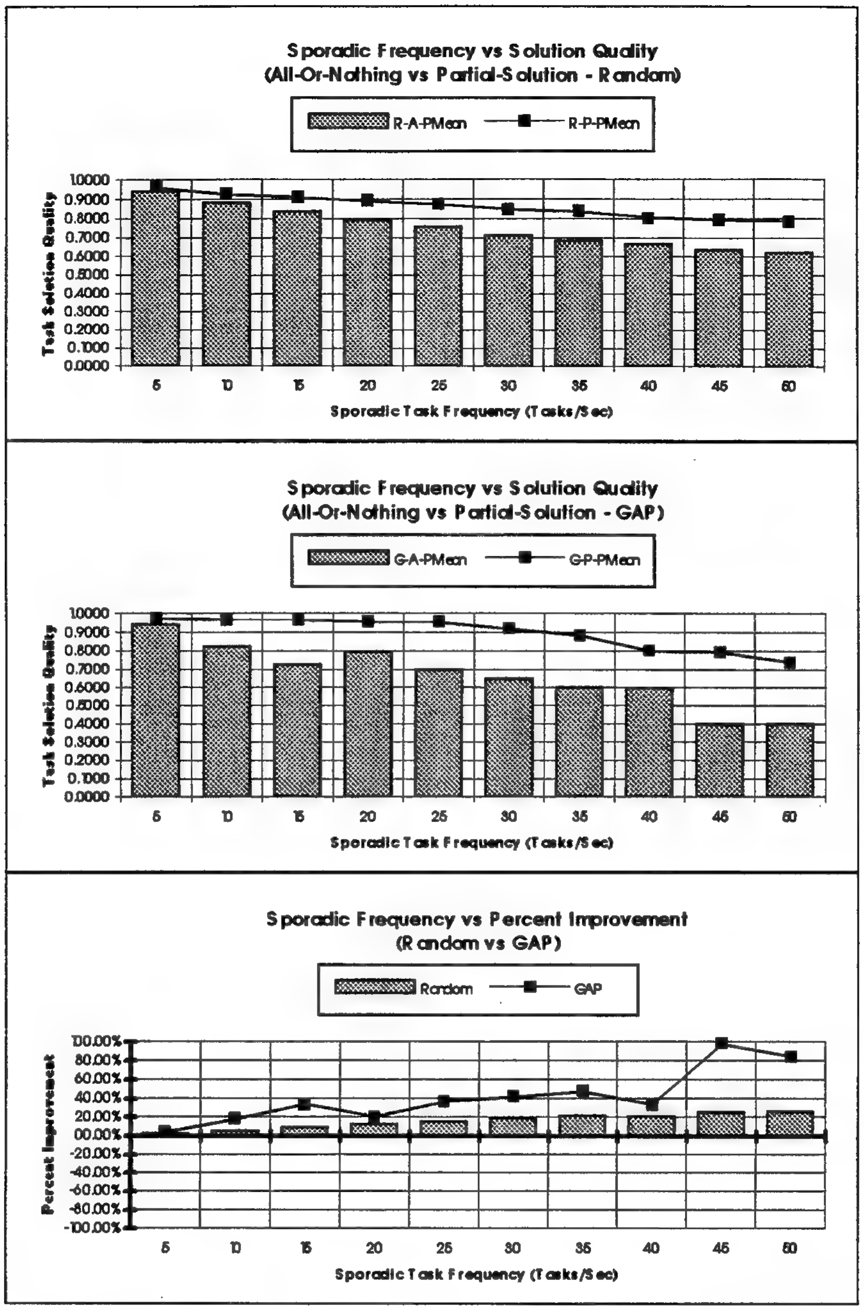


Figure 5.8: Extended Linear-Frequency Metric 1

On the negative side, Figure 5.9 shows that there is a decrease in the solution quality, for the randomly generated Optimistic test, from 20% to 16% when compared with the original test. This result reemphasizes the important relationship between task performance curves and partial-solution performance.

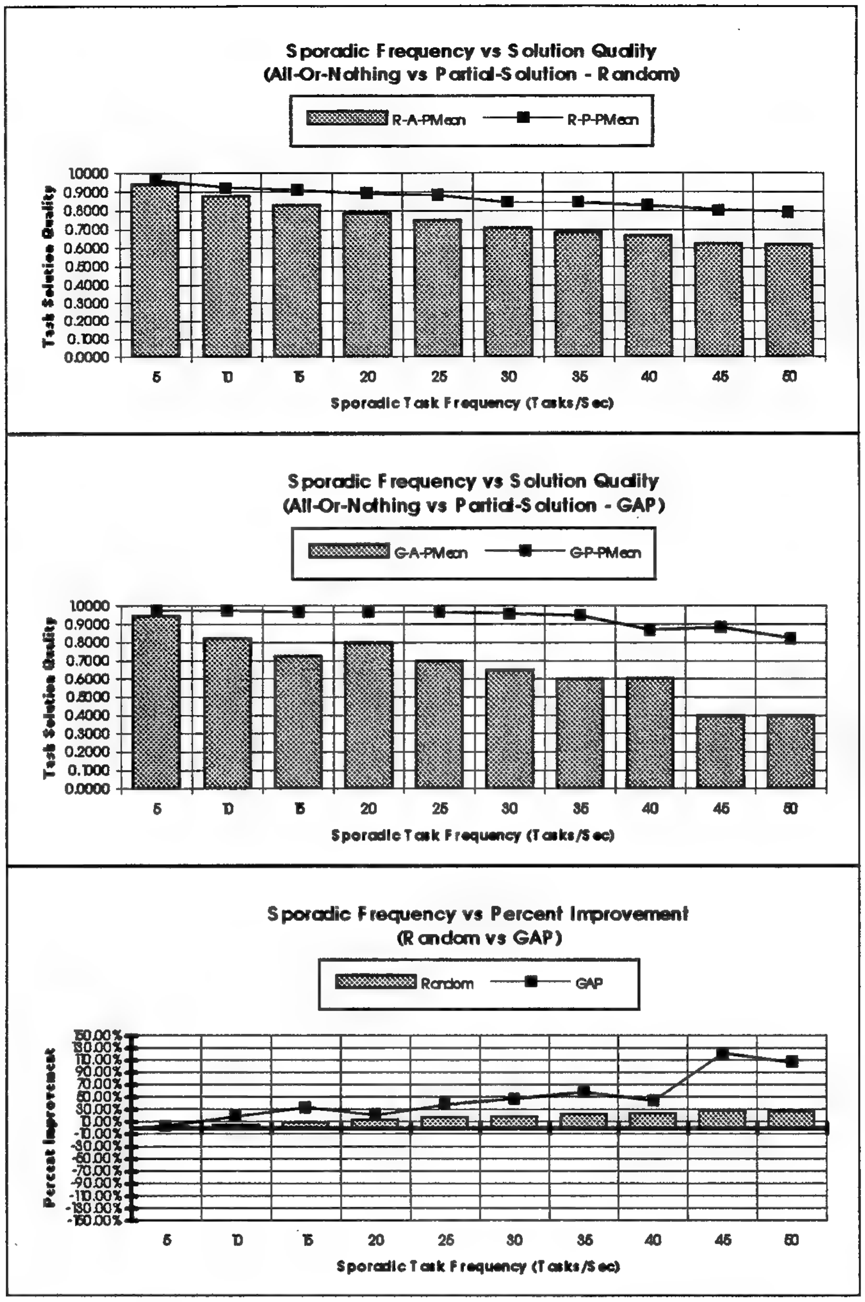


Figure 5.9: Extended Optimistic-Frequency Metric 1

Additional cost can also be seen, in all nine tests, in the increased number of missed cyclic-deadlines (metric 2). However, this is expected, since some of the previous partial-solution tasks are now implemented as all-or-nothing tasks.

5.3 Summary of Results

From a domain-independent perspective, the results presented in this chapter indicate that partial-solution tasks can increase the schedulability, maintainability and robustness of real-time systems.

In the pre-runtime schedulability demonstration, the degraded performance of the partial-solution Graphic-Display allowed the scheduler to generate a feasible schedule when none would have existed using all-or-nothing techniques. As additional tasks were added during the maintainability demonstration, the scheduler continued to generate feasible schedules by continuing to degrade the performance of the Graphic-Display. Only when the utilization requirements exceeded the capability of the partial-solution task was it necessary to add additional resources.

Results from the runtime robustness test appear to be more subjective. If decreasing the number of missed deadlines is the goal, then partial-solution tasks seem quite effective. However, if increasing solution quality is the primary concern, results will depend largely on domain-specific characteristics, such as task performance curves, task durations, and executive resolution.

VI - Conclusion and Recommendations

The primary objective of this research was to show through example and experimentation the benefits associated with developing real-time systems using partial-solution techniques in conjunction with all-or-nothing techniques. In particular, the research focused on demonstrating the potential for increased schedulability, maintainability and robustness. This chapter summarizes the important results presented in Chapter V and provides recommendations for follow-on work.

6.1 Conclusions

The results presented in Chapter V support the research hypothesis that partial-solution tasks can increase the schedulability, maintainability, and robustness of some real-time systems. In particular, the research shows that partial-solution tasks can be used to generate feasible schedules under conditions where all-or-nothing techniques would fail. Furthermore, it shows that partial-solution tasks can increase system maintainability by allowing the scheduler to degrade the performance of the partial-solution tasks in order to accommodate unexpected requirements. During the runtime test it was also shown that partial-solution tasks have the potential for decreasing the number of missed deadlines and increasing a system's average solution quality.

Even though the results presented in this research appear promising, they need to be kept in perspective. Due to a lack of domain-specific requirements and the underlying assumption that tasks are independent, the results may be misleading. For most systems, task dependencies are unavoidable and domain requirements dictate which tasks, if any, can be implemented using partial-solution techniques.

Domain-specific knowledge is also necessary to determine at what level the performance of the partial-solution task becomes unacceptable. For example, it would be unacceptable if the Graphic-Display used in the pre-runtime experiment was unable to update a critical system display with its performance degraded by more than 50%.

Another important result of the research is the correlation between the shape of the performance curve and the impact of the partial-solution task on the performance of the system. The research indicates that tasks achieving most of their quality early during their execution are more likely to lead to increased system performance under unpredictable and dynamic conditions.

Finally, this research has resulted in theory, tools, and techniques for the general case while providing the ability to overlay the specific case. As more domain-specific information becomes available, these theories and tools can be used to generate performance perdictions for particular cases.

6.2 Recommendations for Future Work

One of the objectives of this research was to determine if partial-solution tasks have domain-independent characteristics that allow them to improve the performance of a real-time system. Finding a domain-independent technique could lead to a reduction in the cost and effort required to build next-generation systems, by eliminating the need to gather domain-specific information. However, the research does not provide any major breakthroughs. Rather, it uncovers a need for improved techniques for analyzing and recording the specific characteristics of real-time systems.

John Stankovic points out that there is no need to treat a task as a random process [Stankovic, 1990]. Many characteristics of tasks (such as their timing and resource requirements) can be determined *a priori* and utilized at runtime. Future research should focus some attention on the areas of criticality and acceptable degradation, since it is these characteristics that will ultimately be used by the intelligent scheduler to determine which tasks to schedule and how much to degrade each task.

Due to timing and resource constraints, several key design decisions were made in this research that forced the assumption of an overly simplified real-time environment. First, the unreliability of the UNIX operating system makes it impossible to enforce real-time guarantees in a rigorous fashion. Second, the Silicon Graphics clock resolution makes it impossible to generate the precise timing accuracy required by many real-time tasks. Finally, the randomly generated cyclic schedules, task importances, task durations, and task response times and the assumption of task independence may not reflect the true characteristics of next-generation real-time systems.

Overcoming the reliability of the operating system and improving the clock resolution could be achieved by porting the runtime executive from the UNIX general-purpose operating system to a real-time operating system. A system that has just recently been made available is the Maruti operating system being developed at the University of Maryland [Agrawala, 1994]. Maruti provides a comprehensive set of tools for developing mission-critical real-time systems. The current version of the operating system (2.1) runs on a 486 AT/PC platform and includes a full set of tools to prepare and profile real-time applications. Information on acquiring Maruti can be obtained by sending electronic mail to professor Ashok Agrawala (agrawala@cs.umd.edu).

An alternative approach to improving the existing software would be to combine the efforts of this Institution (AFIT) with additional efforts being conducted at universities throughout the country. Several of these universities have already developed relatively sophisticated architectures for performing intelligent real-time control. By cooperating with these institutions it would allow follow-on research to focus its attention on domain requirements, by minimizing the need to address implementation issues.

Appendix A - Random Task Requirements

A.1 Description

This appendix describes the timing and importance requirements randomly generated for the runtime schedulability and robustness tests.

A.2 Random Task Requirements

Table A.1: Random Task Requirements

	Schedule 0		Schedule 1		Schedule 2		Schedule 3	
Task	Priority	Duration	Priority	Duration	Priority	Duration	Priority	Duration
Task0	N	0.04	E	0.03	C	0.09	E	0.02
Task1	C	0.08	E	0.03	C	0.04	C	0.04
Task2	N	0.02	E	0.06	C	0.08	E	0.10
Task3	E	0.02	C	0.04	N	0.04	C	0.02
Task4	E	0.08	E	0.09	E	0.03	C	0.02
Task5	C	0.08	N	0.02	N	0.06	E	010
Task6	N	0.10	E	0.08	C	0.07	C	0.05
Task7	E	0.10	N	0.07	E	0.03	E	0.02
Task8	E	0.10	E	0.06	E	0.02	E	0.08
Task9	C	0.02	E	0.02	C	0.10	N	0.03

	Schedule 4		Schedule 5		Schedule 6		Schedule 7	
Task	Priority	Duration	Priority	Duration	Priority	Duration	Priority	Duration
Task0	E	0.02	N	0.08	E	0.05	N	0.02
Task1	E	0.02	E	0.07	N	0.10	C	0.09
Task2	E	0.02	C	0.10	C	0.05	N	0.03
Task3	E	0.10	C	0.06	E	0.06	N	0.08
Task4	E	0.02	E	0.05	E	0.06	C	0.09
Task5	E	0.04	C	0.03	E	0.02	E	0.07
Task6	N	0.02	N	0.08	N	0.07	C	0.07
Task7	E	0.02	E	0.05	C	0.07	C	0.02
Task8	E	0.02	N	0.05	E	0.07	E	0.10
Task9	C	0.05	C	0.02	E	0.09	E	0.02

	Schedule 8		Schedule 9		Schedule GAP	
Task	Priority	Duration	Priority	Duration	Priority	Duration
Task0	N	0.08	N	0.02	N	0.00
Task1	C	0.09	N	0.02	E	0.05
Task2	E	0.02	E	0.06	N	0.05
Task3	N	0.06	N	0.02	N	0.08
Task4	C	0.08	C	0.09	C	0.09
Task5	E	0.02	E	0.02	C	0.05
Task6	C	0.07	N	0.08	C	0.01
Task7	E	0.09	N	0.05	N	0.03
Task8	E	0.05	E	0.06	N	0.01
Task9	C	0.02	N	0.10	N	0.03

Appendix B - Pre-Runtime Results

B.1 Description

This appendix presents results from the pre-runtime schedulability and maintenance demonstration. Results are presented in chronological order and were generated using the pre-runtime schedulability software described in Chapter IV. A written summary of the results can be found in Chapter V.

B.2 Original GAP Schedule

```
> (print-schedule 1 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13)
NIL
```

Figure B.1: Infeasible GAP Schedule

B.3 GAP Schedule with Modified Graphic-Display

```
-> (print-schedule 1 t1 t2 t3 t4m t5 t6 t7 t8 t9 t10 t11 t12 t13)
-----------------------------------------------------------------
Processor #1

     Name               Period  Duration  Priority  Importance  Processor
     ----------------   ------  --------  --------  ----------  ---------
     KEYSET             0.2     0.001     38        3           1
     GRAPHIC_DISPLAY    0.08    0.005     36        3           1
     TARGET_UPDATE_II   0.1     0.005     37        3           1
     CONTACT_MGNT       0.025   0.005     31        2           1
     STATUS_UPDATE      0.2     0.003     41        1           1
     HOOK_UPDATE        0.065   0.002     35        1           1
     STORES_UPDATE      0.2     0.001     40        1           1
     TARGET_UPDATE_I    0.05    0.005     33        1           1
     TRACKING_FILTER    0.025   0.002     32        1           1
     NAV_UPDATE         0.059   0.008     34        1           1
     STEERING_CMDS      0.2     0.003     39        1           1
     NAV_STATUS         1.0     0.001     42        1           1
     STATUS_UPDATE_II   1.0     0.001     43        1           1

     Util      [0.7008624511082139]
     Max Util  [0.7119589942614066]
```

Figure B.2: GAP Schedule with Modified Graphic-Display

B.4 GAP Schedule with Modified Graphic-Display and Maintainance Requirements

```
> (print-schedule 1 t1 t2 t3 t4m t5 t6 t7 t8 t9 t10 t11 t12 t13 t1-extra)
--------------------------------------------------------------
Processor #1

     Name                Period   Duration   Priority   Importance   Processor
     ----------------    ------   --------   --------   ----------   ---------
     KEYSET              0.2      0.001      38         3            1
     GRAPHIC_DISPLAY     0.08     0.005      36         3            1
     TARGET_UPDATE_II    0.1      0.005      37         3            1
     CONTACT_MGNT        0.025    0.005      31         2            1
     STATUS_UPDATE       0.2      0.003      39         1            1
     HOOK_UPDATE         0.065    0.002      35         1            1
     STORES_UPDATE       0.2      0.001      40         1            1
     TARGET_UPDATE_I     0.05     0.005      33         1            1
     TRACKING_FILTER     0.025    0.002      32         1            1
     NAV_UPDATE          0.059    0.008      34         1            1
     STEERING_CMDS       0.2      0.003      41         1            1
     NAV_STATUS          1.0      0.001      43         1            1
     STATUS_UPDATE_II    1.0      0.001      44         1            1
     COLLISION_MONITOR   0.2      0.0003     42         1            1

     Util     [0.7023624511082137]
     Max Util [0.7105929411450722]
```

Figure B.3: Modified GAP with One Maintenance Task

```
> (print-schedule 1 t1 t2 t3 t4m t5 t6 t7 t8 t9 t10 t11 t12 t13
                  t1-extra t2-extra)
--------------------------------------------------------------
Processor #1

     Name                Period   Duration   Priority   Importance   Processor
     ----------------    ------   --------   --------   ----------   ---------
     KEYSET              0.2      0.001      38         3            1
     GRAPHIC_DISPLAY     0.08     0.004      36         3            1
     TARGET_UPDATE_II    0.1      0.005      37         3            1
     CONTACT_MGNT        0.025    0.005      31         2            1
     STATUS_UPDATE       0.2      0.003      39         1            1
     HOOK_UPDATE         0.065    0.002      35         1            1
     STORES_UPDATE       0.2      0.001      40         1            1
     TARGET_UPDATE_I     0.05     0.005      33         1            1
     TRACKING_FILTER     0.025    0.002      32         1            1
     NAV_UPDATE          0.059    0.008      34         1            1
     STEERING_CMDS       0.2      0.003      41         1            1
     NAV_STATUS          1.0      0.001      43         1            1
     STATUS_UPDATE_II    1.0      0.001      44         1            1
     COLLISION_MONITOR   0.2      0.0003     42         1            1
     PILOT_MANAGER       1.0      0.0081     45         1            1

     Util     [0.6979624511082138]
     Max Util [0.7094118423094009]
```

Figure B.4: Modified GAP with Two Maintenance Tasks

```
> (print-schedule 1 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13
                    t1-extra t2-extra t3-extra)
---------------------------------------------------------------
Processor #1

     Name                Period  Duration  Priority  Importance  Processor
     ----------------    ------  --------  --------  ----------  ---------
     KEYSET              0.2     0.001     38        3           1
     GRAPHIC_DISPLAY     0.08    0.002     36        3           1
     TARGET_UPDATE_II    0.1     0.005     37        3           1
     CONTACT_MGNT        0.025   0.005     31        2           1
     STATUS_UPDATE       0.2     0.003     39        1           1
     HOOK_UPDATE         0.065   0.002     35        1           1
     STORES_UPDATE       0.2     0.001     40        1           1
     TARGET_UPDATE_I     0.05    0.005     33        1           1
     TRACKING_FILTER     0.025   0.002     32        1           1
     NAV_UPDATE          0.059   0.008     34        1           1
     STEERING_CMDS       0.2     0.003     41        1           1
     NAV_STATUS          1.0     0.001     43        1           1
     STATUS_UPDATE_II    1.0     0.001     44        1           1
     COLLISION_MONITOR   0.2     0.0003    42        1           1
     PILOT_MANAGER       1.0     0.0081    45        1           1
     TACTICAL_MANAGER    1.0     0.0269    46        1           1

     Util     [0.6998624511082138]
     Max Util [0.7083805188386201]
```

Figure B.5: Modified GAP with Three Maintenance Tasks

```
> (print-schedule 1 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13
                    t1-extra t2-extra t3-extra t4-extra)
NIL
```

Figure B.6: Modified GAP with Four Maintenance Tasks and One Processor

```
> (print-schedule 2 t1 t2 t3 t4m t5 t6 t7 t8 t9 t10 t11 t12 t13
                    t1-extra t2-extra t3-extra t4-extra)
---------------------------------------------------------------
Processor #1

     Name                Period  Duration  Priority  Importance  Processor
     ----------------    ------  --------  --------  ----------  ---------
     NAV_UPDATE          0.059   0.008     31        1           1
     EOB_MANAGER         1.0     0.1331    32        1           1

     Util     [0.268693220338983]
     Max Util [0.8284271247461899]
---------------------------------------------------------------
Processor #2

     Name                Period  Duration  Priority  Importance  Processor
     ----------------    ------  --------  --------  ----------  ---------
     KEYSET              0.2     0.001     37        3           2
     GRAPHIC_DISPLAY     0.08    0.009     35        3           2
     TARGET_UPDATE_II    0.1     0.005     36        3           2
     CONTACT_MGNT        0.025   0.005     31        2           2
     STATUS_UPDATE       0.2     0.003     38        1           2
     HOOK_UPDATE         0.065   0.002     34        1           2
     STORES_UPDATE       0.2     0.001     39        1           2
     TARGET_UPDATE_I     0.05    0.005     33        1           2
     TRACKING_FILTER     0.025   0.002     32        1           2
     STEERING_CMDS       0.2     0.003     40        1           2
     NAV_STATUS          1.0     0.001     42        1           2
     STATUS_UPDATE_II    1.0     0.001     43        1           2
     COLLISION_MONITOR   0.2     0.0003    41        1           2
     PILOT_MANAGER       1.0     0.0081    44        1           2
     TACTICAL_MANAGER    1.0     0.0269    45        1           2

     Util     [0.6517692307692307]
     Max Util [0.7094118423094009]
```

Figure B.7: Modified GAP with Four Maintenance Tasks and Two Processors

Appendix C - Initial Runtime Results

C.1 Description

This appendix presents tabular and graphical results from the scheduled phase of the runtime schedulability and robustness test. These results, which are referred to as the *initial results*, were gathered by experimentally measuring the impact of partial-solution tasks on eleven simulated real-time systems (schedules), each of which contained ten cyclic tasks. In total, nine tests were run on each simulated systems:

- Conservative Performance - Fixed Frequency.Test
- Conservative Performance - Fixed Duration Test
- Conservative Performance - Fixed Response Test
- Linear Performance - Fixed Frequency Test
- Linear Performance - Fixed Duration Test
- Linear Performance - Fixed Response Test
- Optimistic Performance - Fixed Frequency Test
- Optimistic Performance - Fixed Duration Test
- Optimistic Performance - Fixed Response Test

Data for all-or-nothing tasks was collected by running the cyclic executive for 20 cycles with all cyclic tasks implemented as all-or-nothing tasks. Data for the partial-solution tasks was collected in the same manner as the all-or-nothing tasks, only in this situation with all cyclic tasks implemented as partial-solution tasks.

Results presented in this section have been classified by task performance curve, test type, metric, and tasking requirement. GAP results were calculated by averaging the performance data over one GAP schedule of 10 cyclic tasks. Random results were calculated by averaging the performance data over the 10 random schedules of 10 cyclic tasks.

A short description of terms used in these results is provided in Table C.1. A written summary of the results can be found in Chapter V.

Table C.1: Key

Metric 1	Average solution quality for cyclic tasks
Metric 2	Percent of missed deadlines for cyclic tasks
Metric 3	Average solution quality for sporadic tasks
Metric 4	Percent of missed deadlines for sporadic tasks
+	Improved quality or decreased missed deadlines
~	No change in quality or missed deadlines
-	Degraded quality or increased missed deadlines
R	Random task requirements
G	GAP task requirements
B	Both random and GAP task requirements
R-A-PMean	Mean value for cyclic all-or-nothing random tasks
R-P-PMean	Mean value for cyclic partial-solution random tasks
G-A-PMean	Mean value for cyclic all-or-nothing GAP tasks
G-P-PMean	Mean value for cyclic partial-solution GAP tasks

C.2 Tabular Results

Table C.2: Runtime Results

		Cyclic-Quality (metric 1)			Cyclic-Deadlines (metric 2)			Sporadic-Quality (metric 3)			Sporadic-Deadlines (metric 4)		
Curve	Test	+	~	-	+	~	-	+	~	-	+	~	-
Conservative	Duration			B	B				B			B	
	Frequency			B	B				B			B	
	Response			B	B				B			B	
Linear	Duration			B	B				B			B	
	Frequency		B		B				B			B	
	Response		B		B				B			B	
Optimistic	Duration	R	G		B				B			B	
	Frequency	B			B				B			B	
	Response	B			B				B			B	
	Random Total	3	2	4	9	0	0	0	9	0	0	9	0
	GAP Total	2	3	4	9	0	0	0	9	0	0	9	0

C.3 Conservative Results - Metric 1

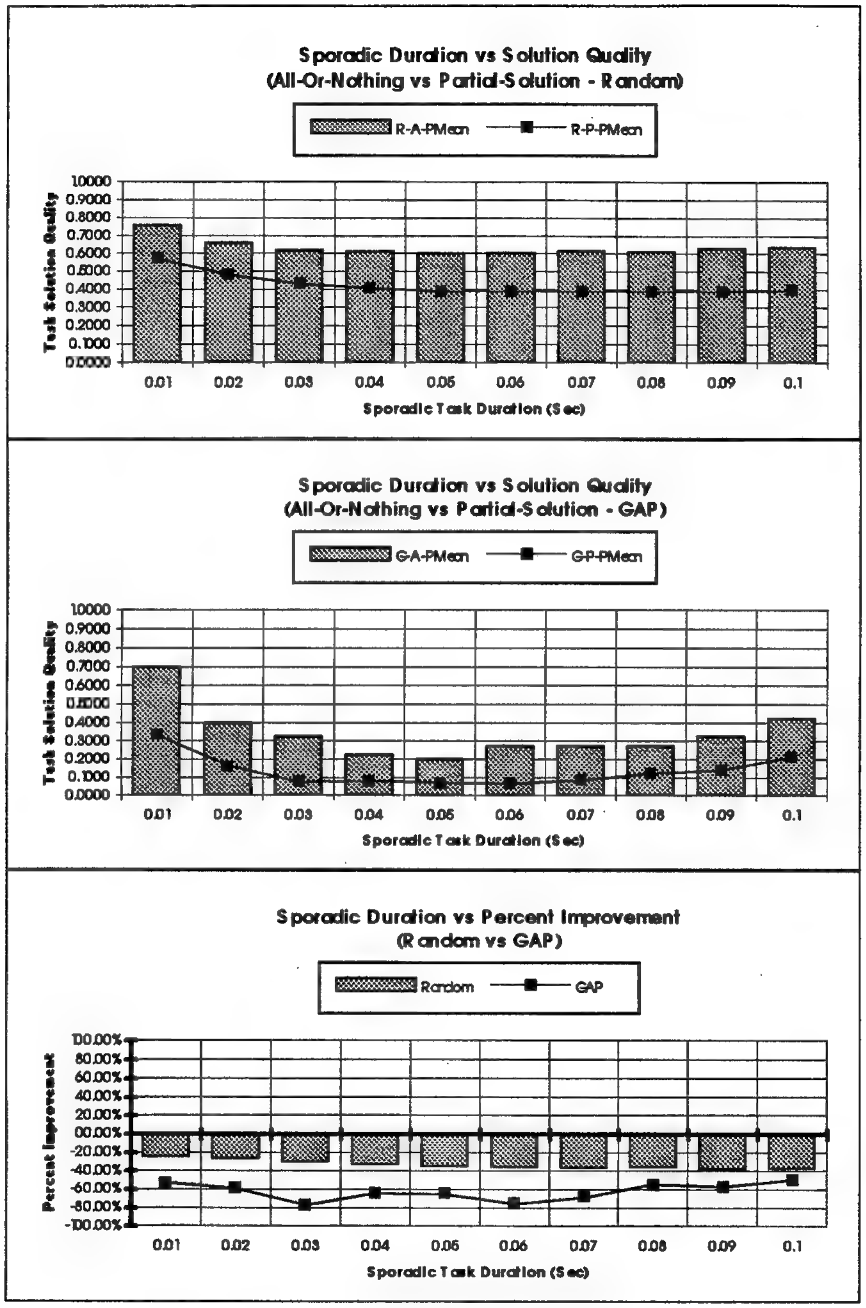


Figure C.1: Conservative-Duration Metric 1

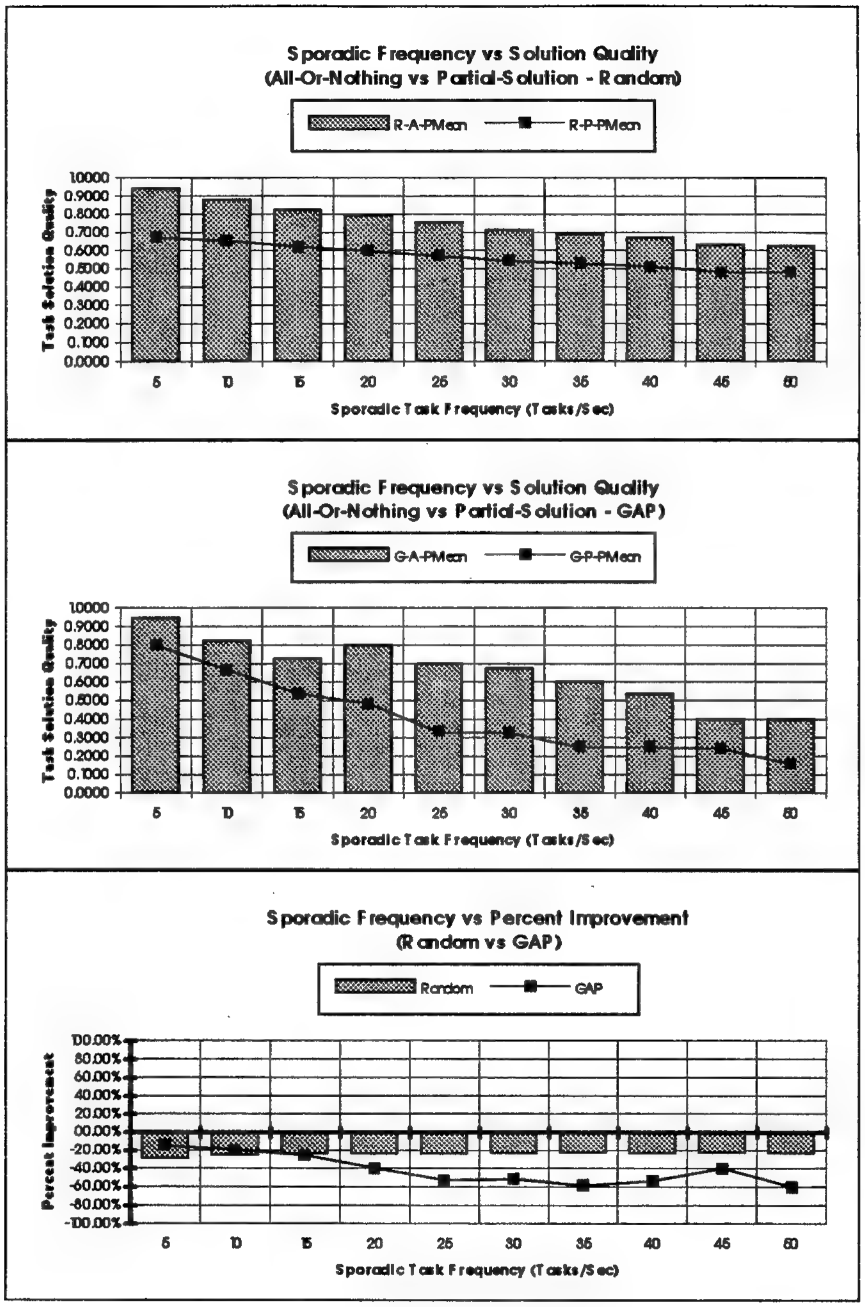
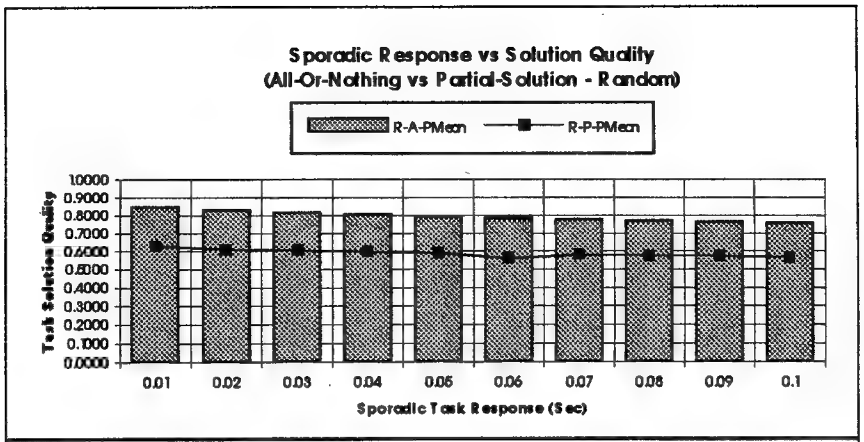
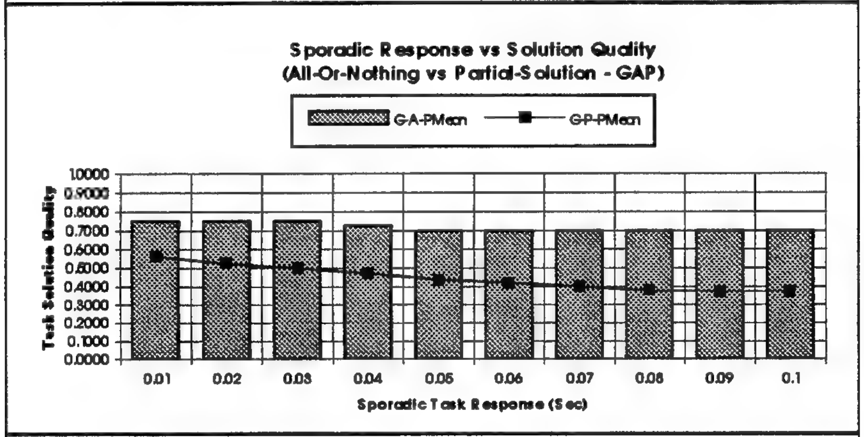


Figure C.2: Conservative-Frequency Metric 1





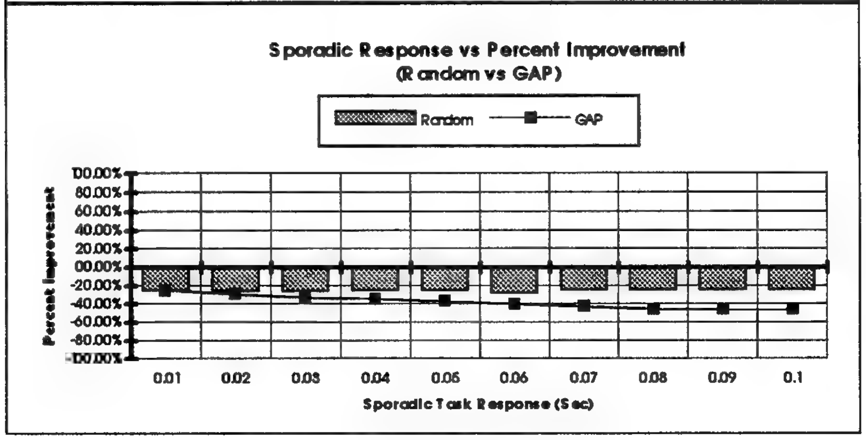


Figure C.3: Conservative-Response Metric 1

C.4 Conservative Results - Metric 2

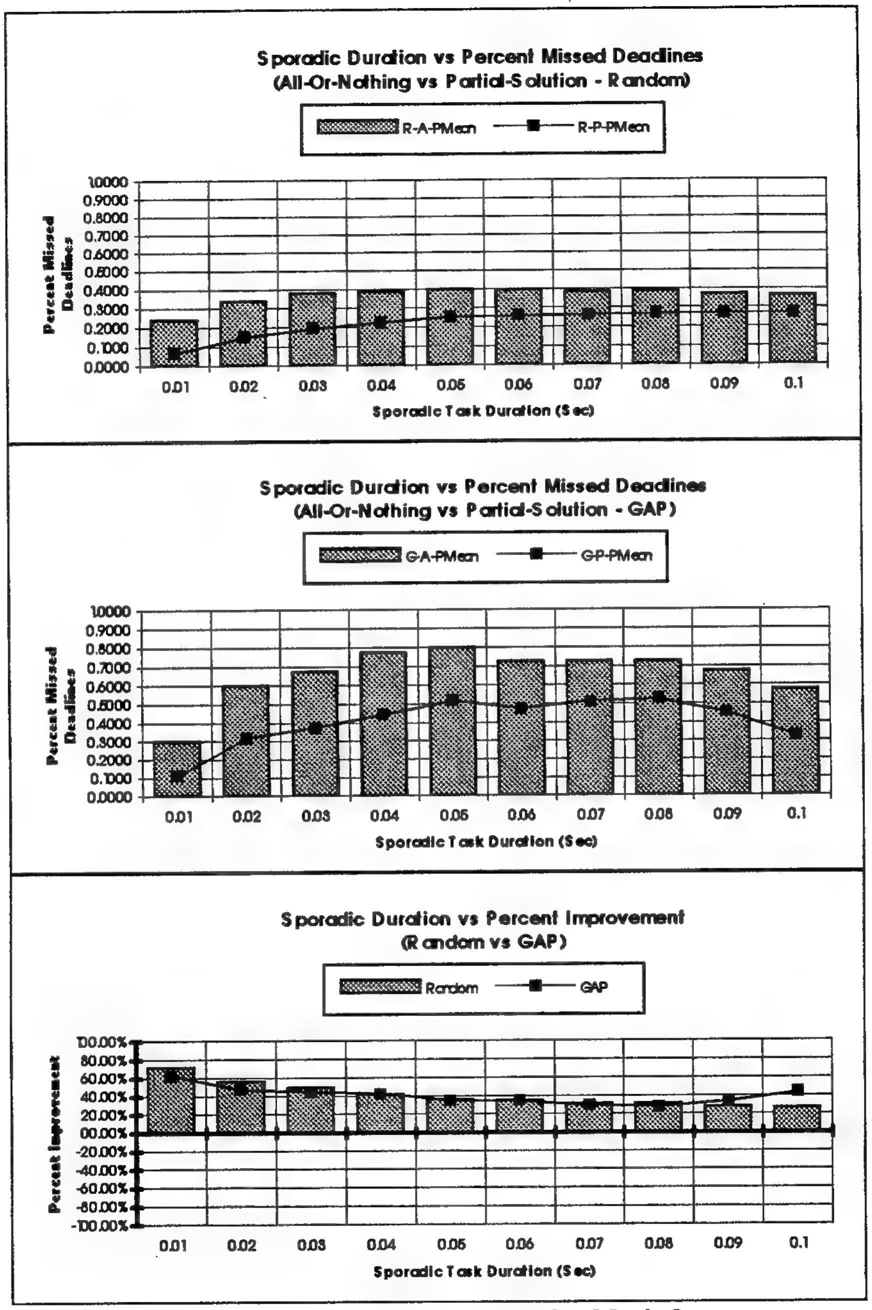


Figure C.4: Conservative-Duration Metric 2

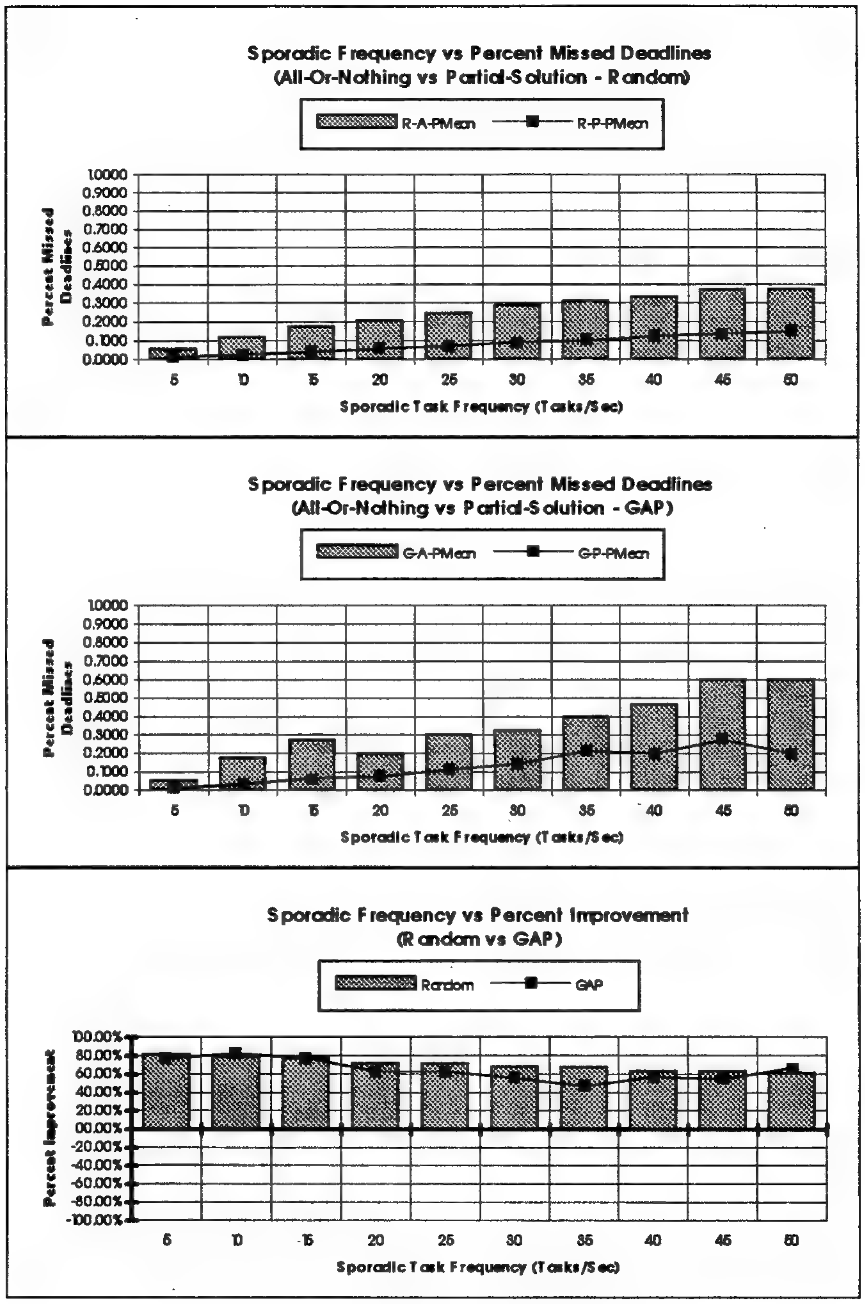


Figure C.5: Conservative-Frequency Metric 2

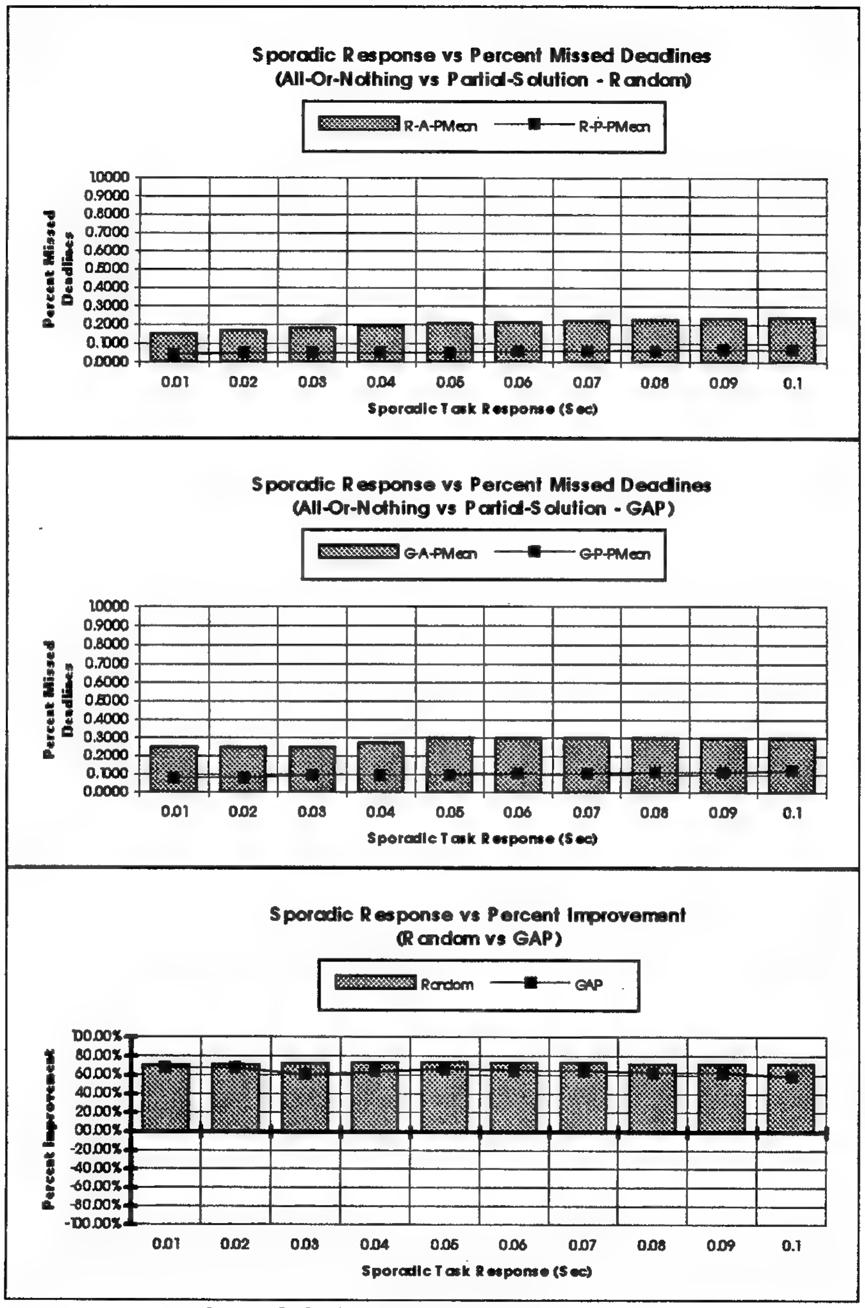
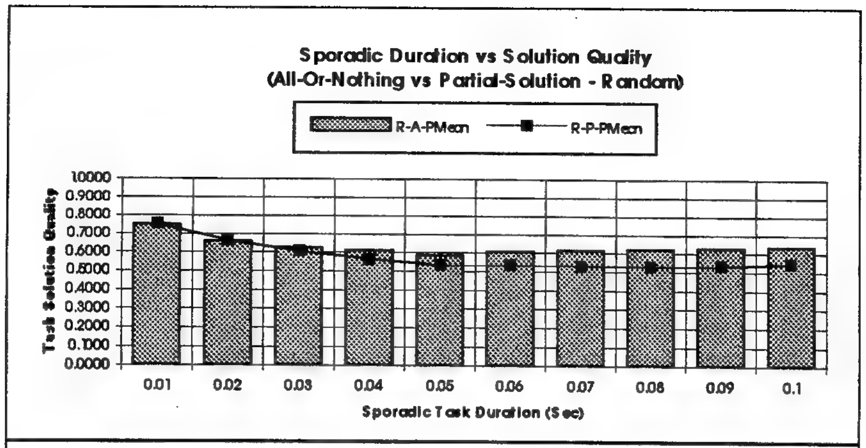
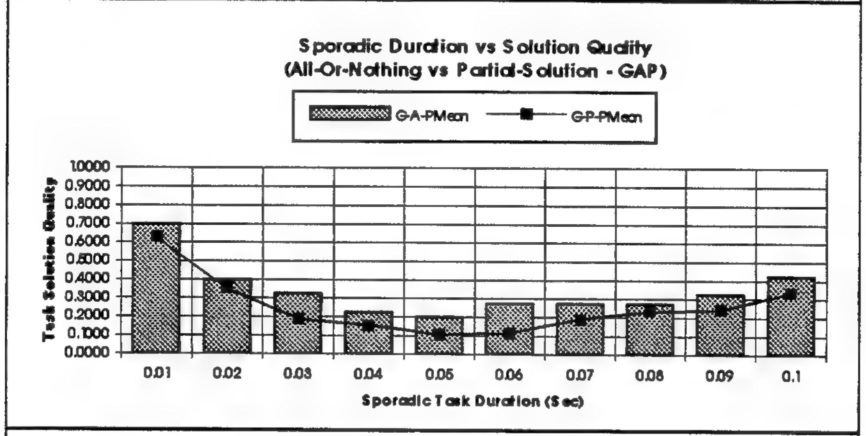


Figure C.6: Conservative-Response Metric 2

C.5 Linear Results - Metric 1





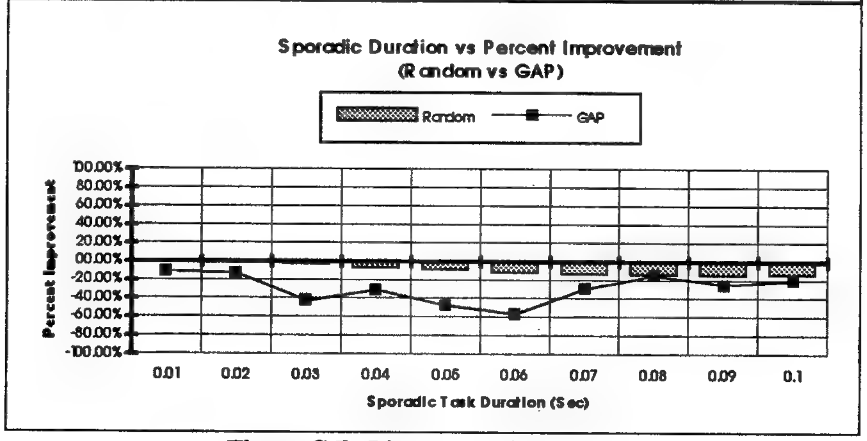


Figure C.7: Linear-Duration Metric 1

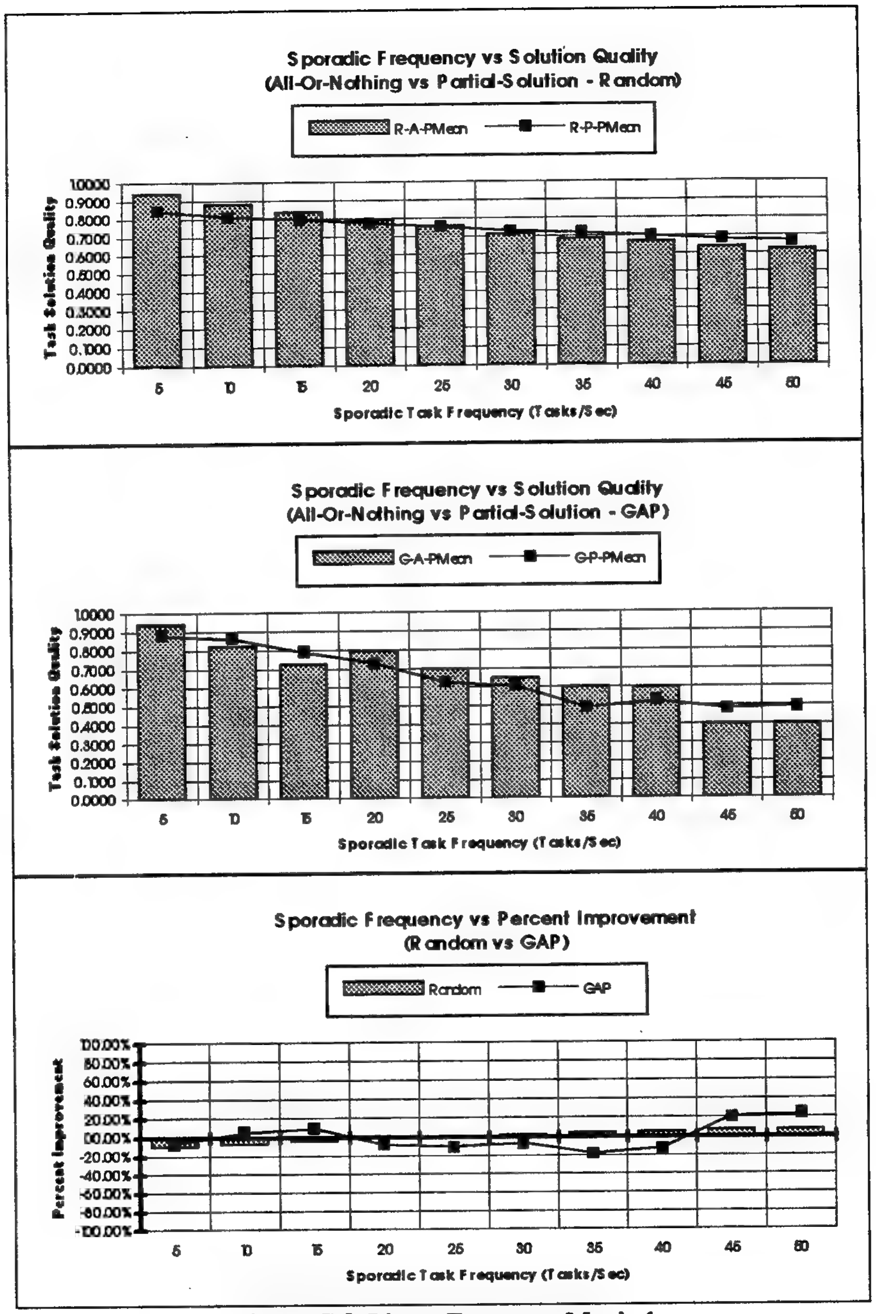


Figure C.8: Linear-Frequency Metric 1

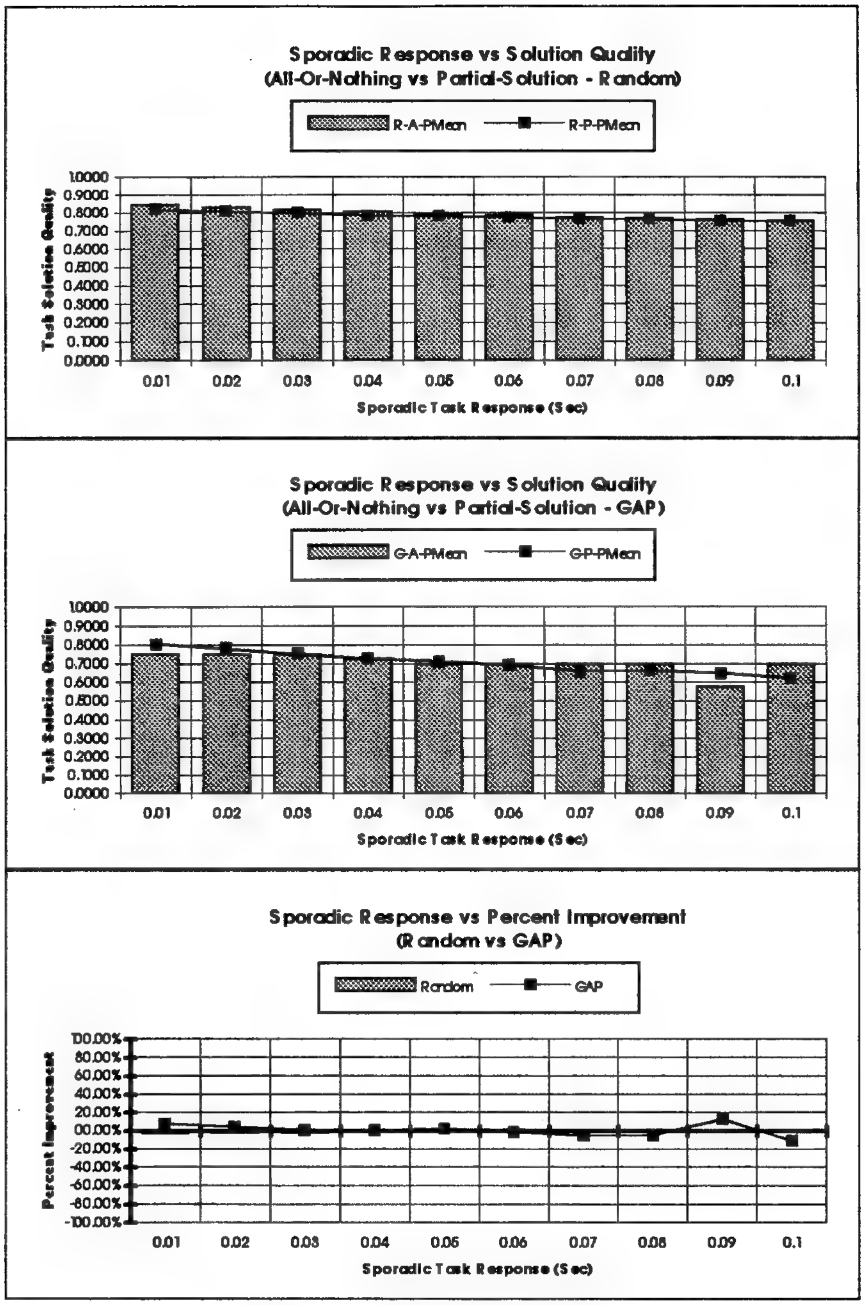


Figure C.9: Linear-Response Metric 1

C.6 Linear Results - Metric 2

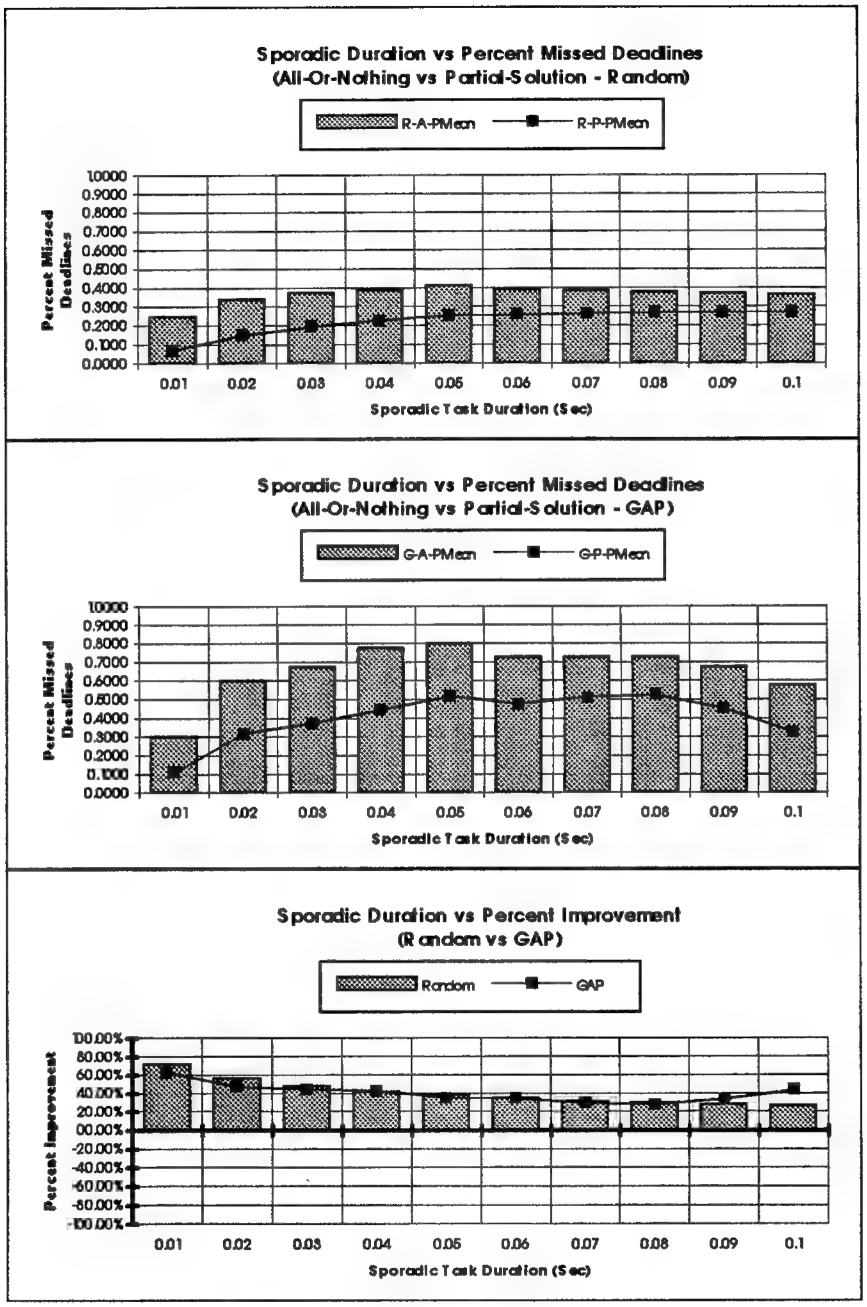


Figure C.10: Linear-Duration Metric 2

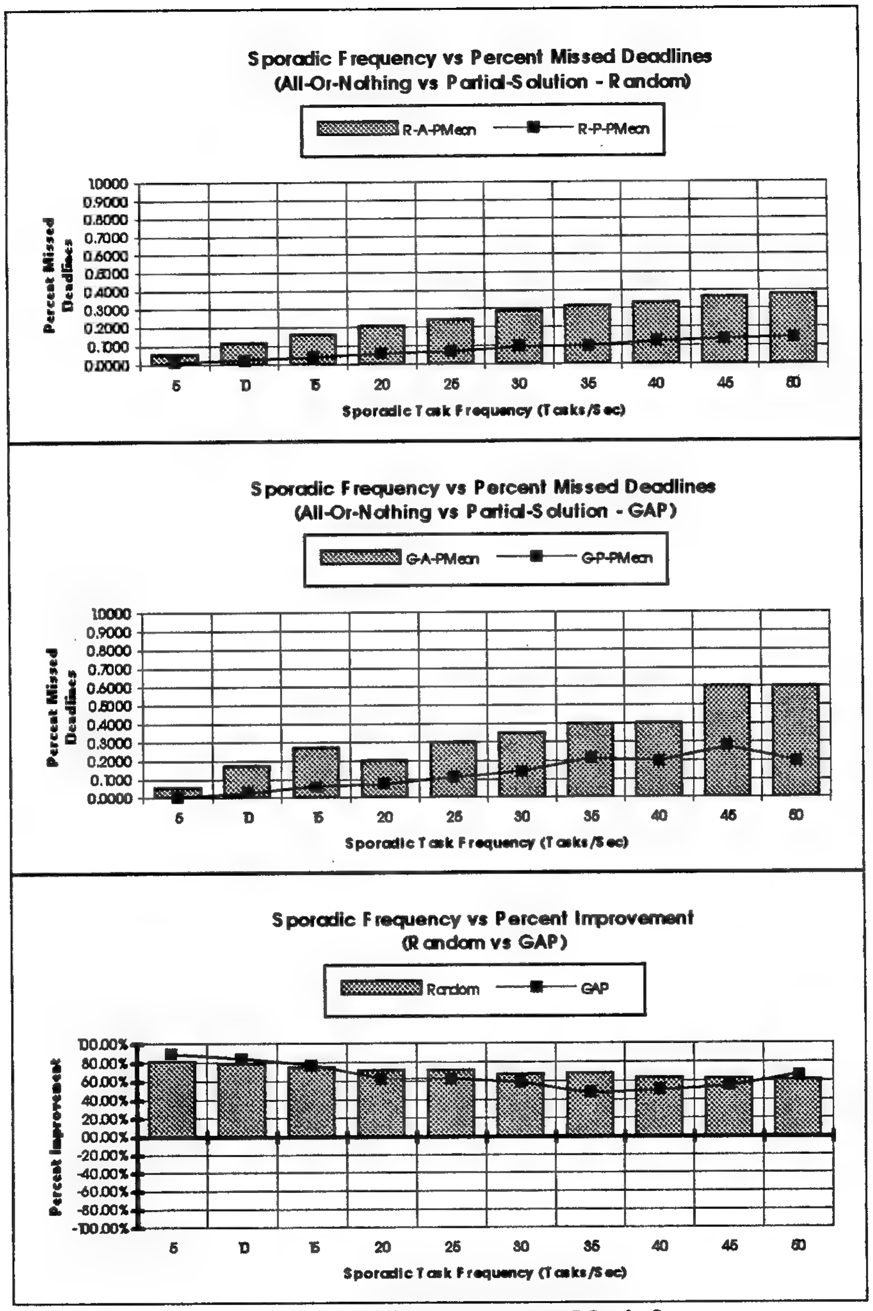
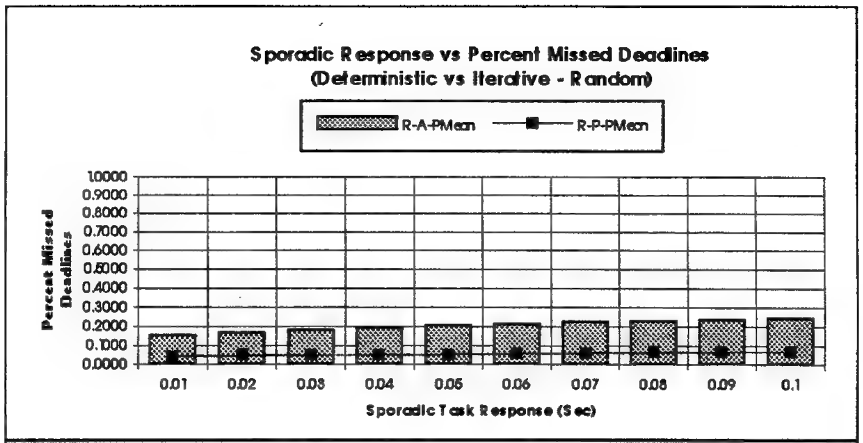
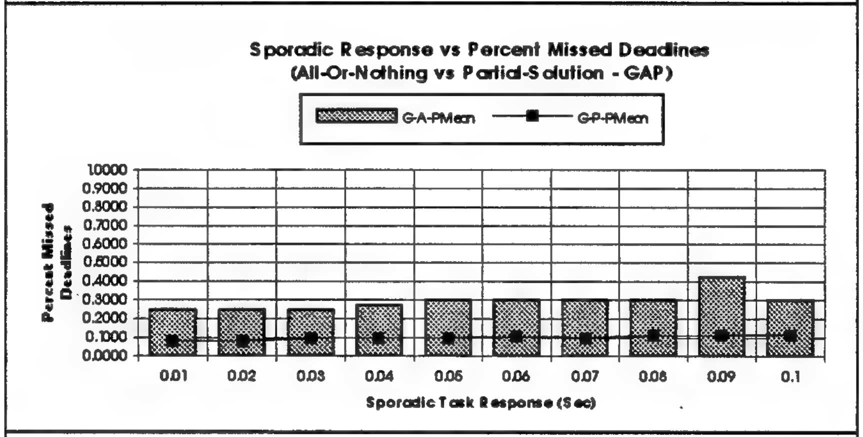


Figure C.11: Linear-Frequency Metric 2





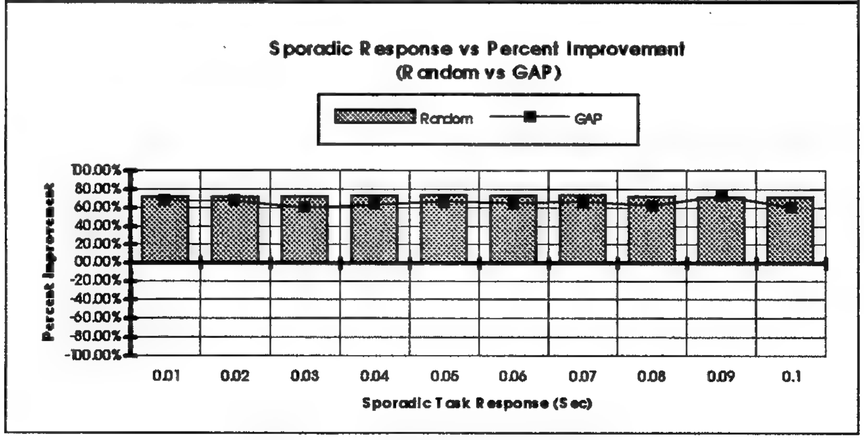


Figure C.12: Linear-Response Metric 2

C.7 Optimistic Results - Metric 1

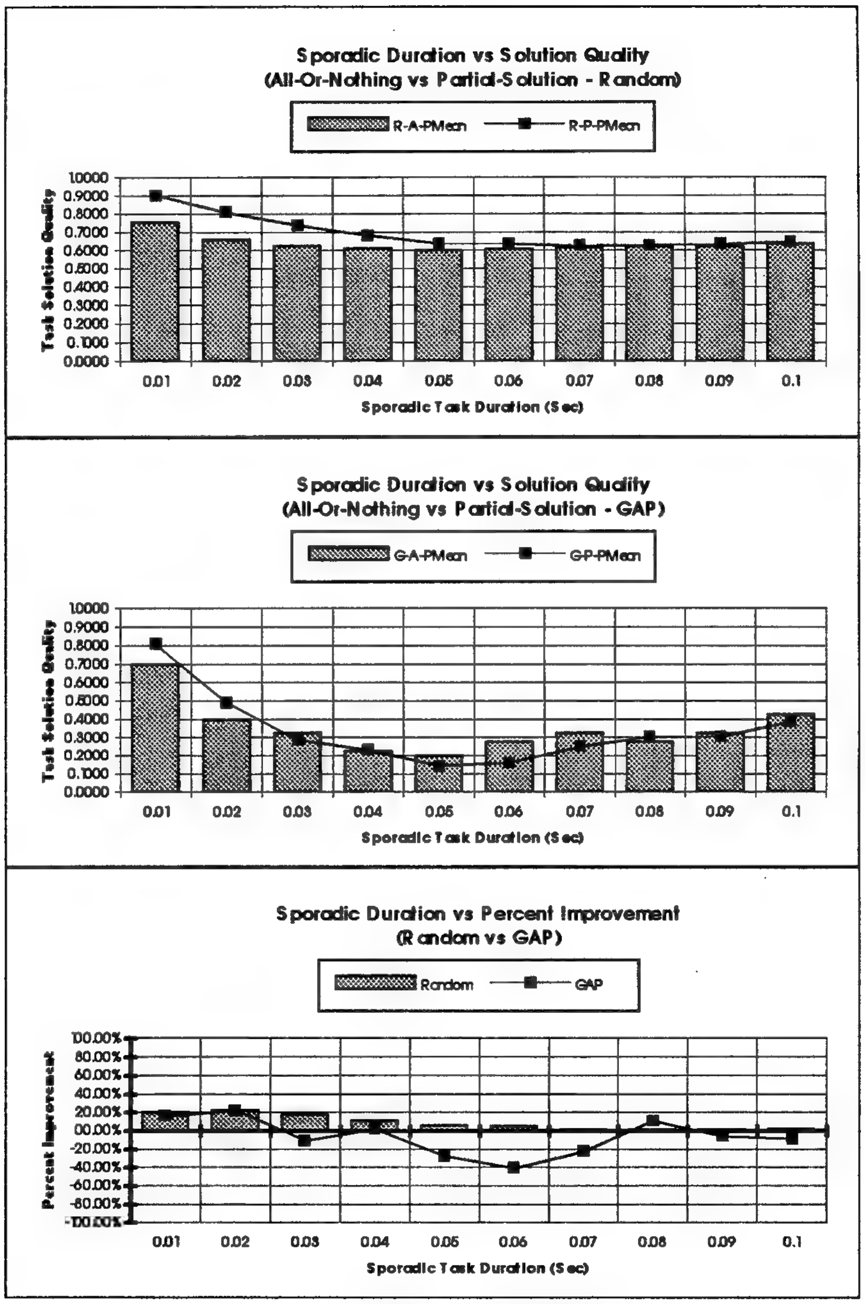


Figure C.13: Optimistic-Duration Metric 1

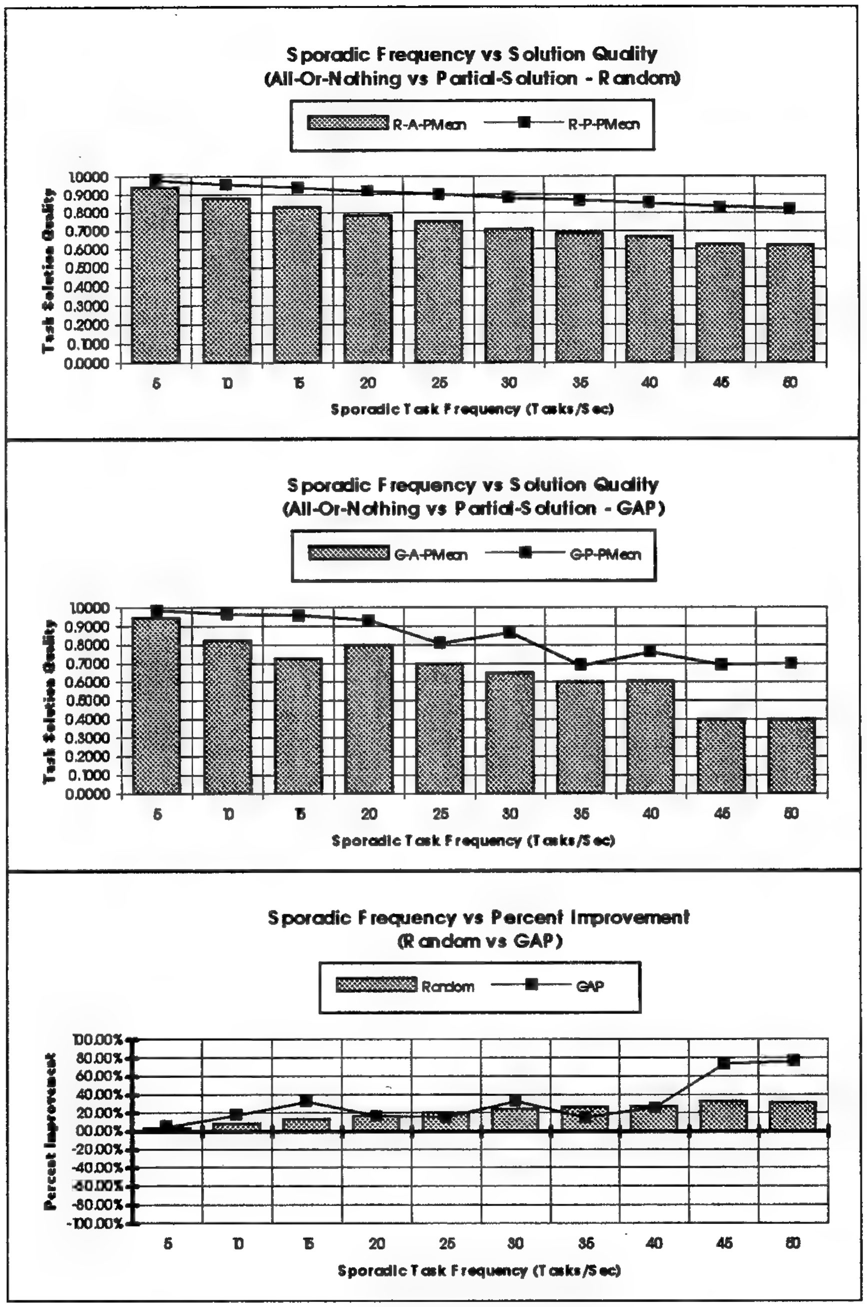


Figure C.14: Optimistic-Frequency Metric 1

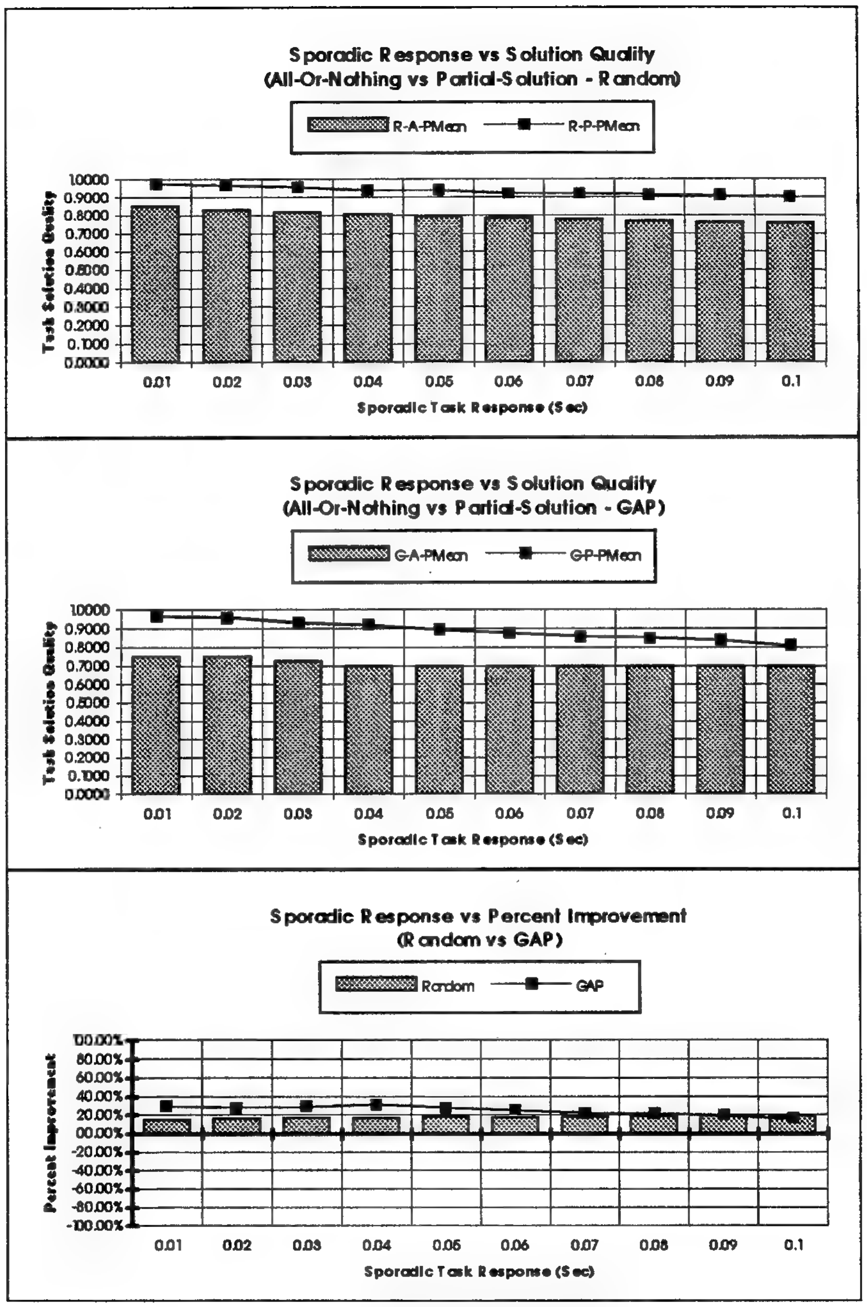
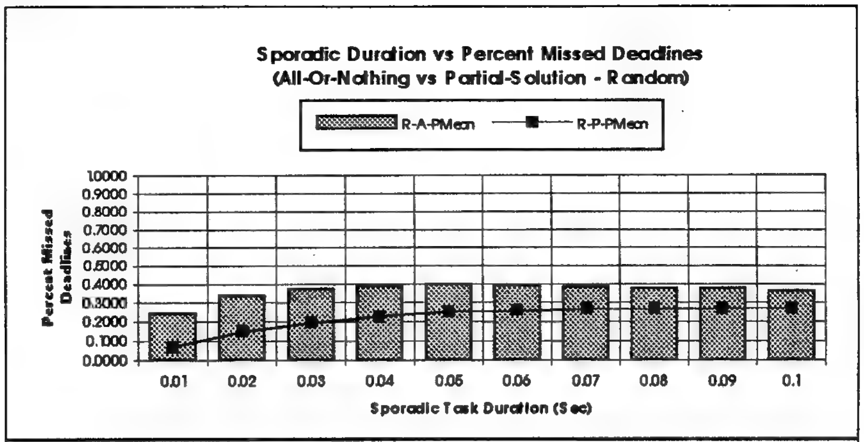
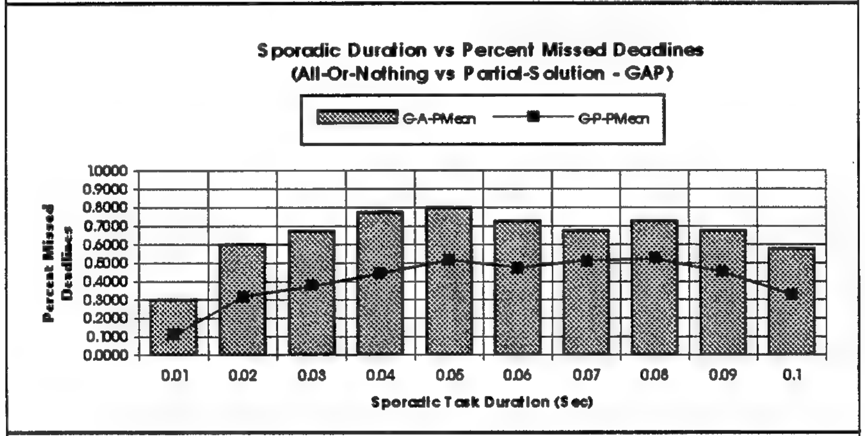


Figure C.15: Optimistic-Response Metric 1

C.8 Optimistic Results - Metric 2





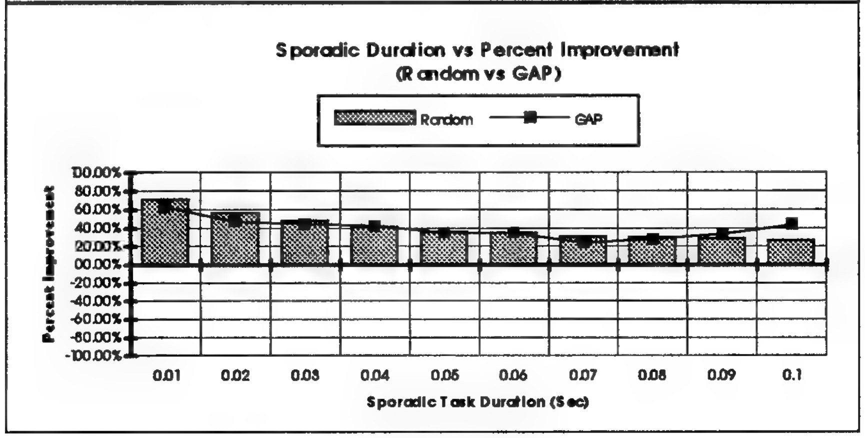


Figure C.16: Optimistic-Duration Metric 2

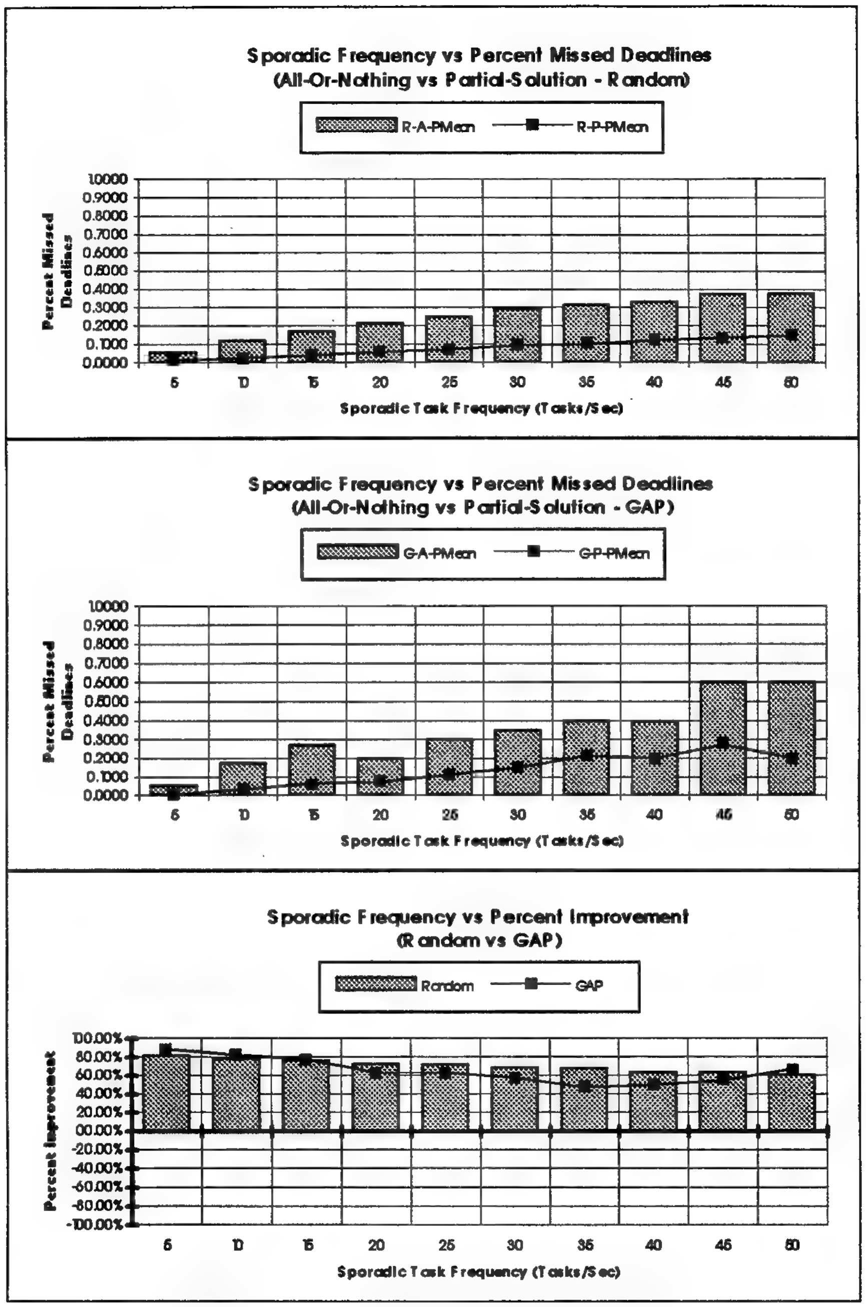
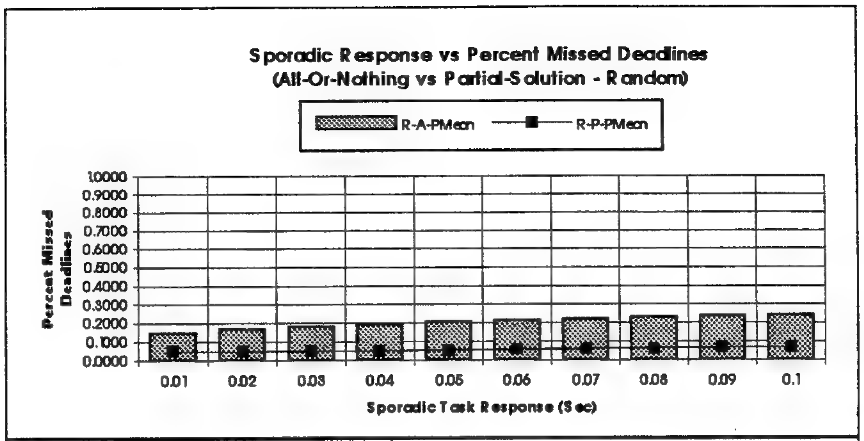
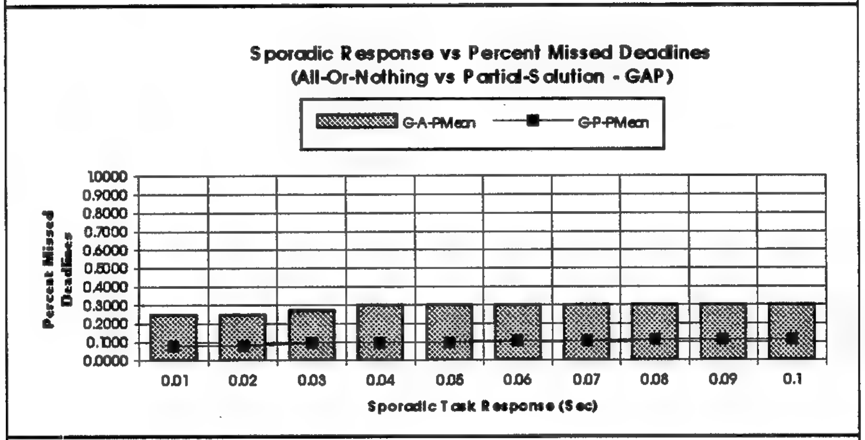


Figure C.17: Optimistic-Frequency Metric 2





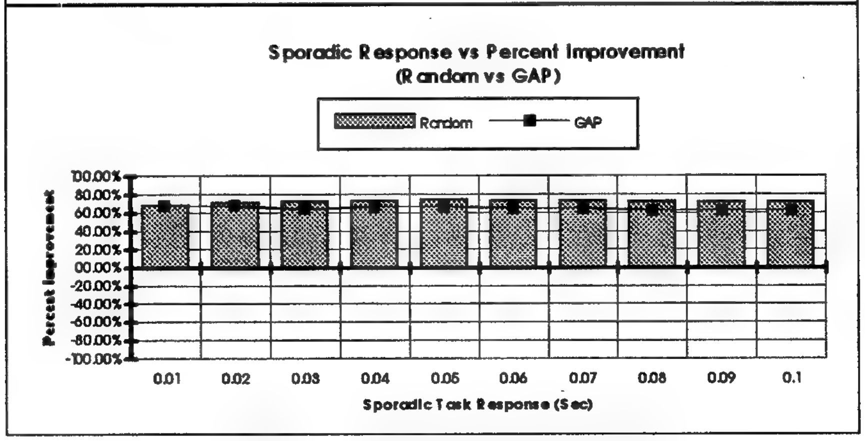


Figure C.18: Optimistic-Response Metric 2

Appendix D - Extended Runtime Results

D.1 Description

This appendix presents tabular and graphical results from the unscheduled phase of the runtime schedulability and robustness test. These results, which are referred to as the *extended results*, were gathered in response to unexpected results during the scheduled (initial) phase of the runtime schedulability and robustness test. The process used for gathering these results was identical to the process used during the scheduled phase, with one exception. In the scheduled phase, all cyclic tasks were modeled as partial-solution tasks. In the unscheduled phase, only cylic tasks with durations of 0.05 seconds or greater were modeled as partial-solution tasks, all others were modeled as all-or-nothing tasks. This modification was made to determine the impact of a small amount of domain-knowledge (task duration and executive resolution) on the performance of the partial-solution tasks.

Table D.1: Key

Metric 1	Average solution quality for cyclic tasks
Metric 2	Percent of missed deadlines for cyclic tasks
Metric 3	Average solution quality for sporadic tasks
Metric 4	Percent of missed deadlines for sporadic tasks
+	Improved quality or decreased missed deadlines
~	No change in quality or missed deadlines
-	Degraded quality or increased missed deadlines
R	Random task requirements
G	GAP task requirements
B	Both random and GAP task requirements
R-A-PMean	Mean value for cyclic all-or-nothing random tasks
R-P-PMean	Mean value for cyclic partial-solution random tasks
G-A-PMean	Mean value for cyclic all-or-nothing GAP tasks
G-P-PMean	Mean value for cyclic partial-solution GAP tasks

Results in this section are classified by task performance curve, test type, metric, and tasking requirement. A short description of terms used in these results is provided in Table D.1. A written summary of the results can be found in Chapter V.

D.2 Tabular Results

Table D.2: Extended Runtime Results

		Cyclic-Quality (metric 1)			Cyclic-Deadlines (metric 2)			Sporadic-Quality (metric 3)			Sporadic-Deadlines (metric 4)		
Curve	Test	+	~	-	+	~	-	+	~	-	+	~	-
Conservative	Duration		B		B				B			B	
	Frequency		B		B				B			B	
	Response		B		B				B			B	
Linear	Duration	B			B				B			B	
	Frequency	B			B				B			B	
	Response	B			B				B			B	
Optimistic	Duration	B			B				B			B	
	Frequency	B			B				B			B	
	Response	B			B				B			B	
	Random Total	6	3	0	9	0	0	0	9	0	0	9	0
	GAP Total	6	3	0	9	0	0	0	9	0	0	9	0

THIS SPACE INTENTIONALLY LEFT BLANK

D.3 Conservative Results - Metric 1

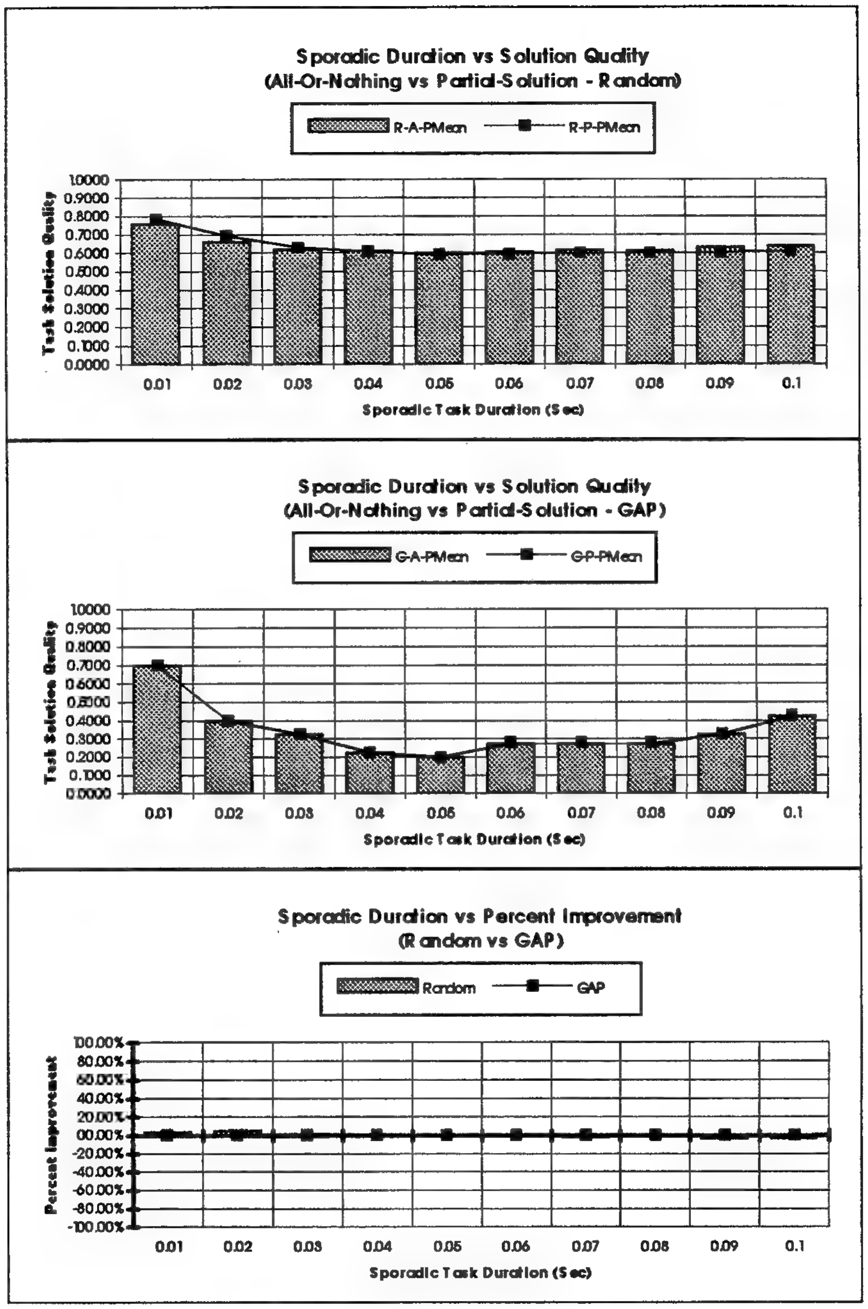
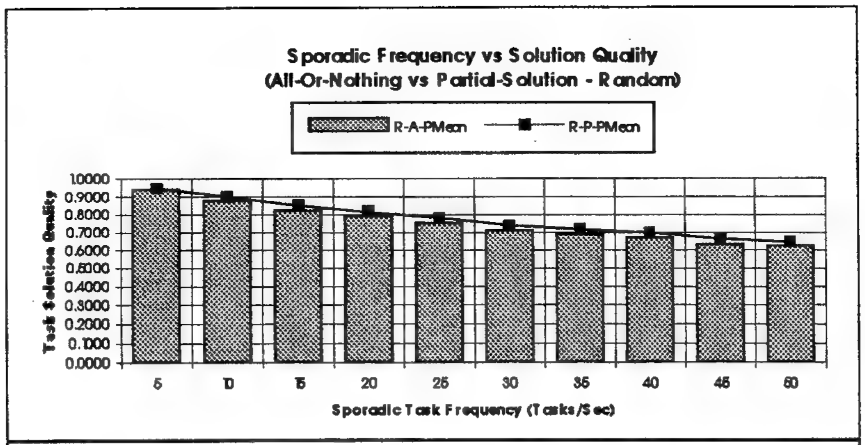
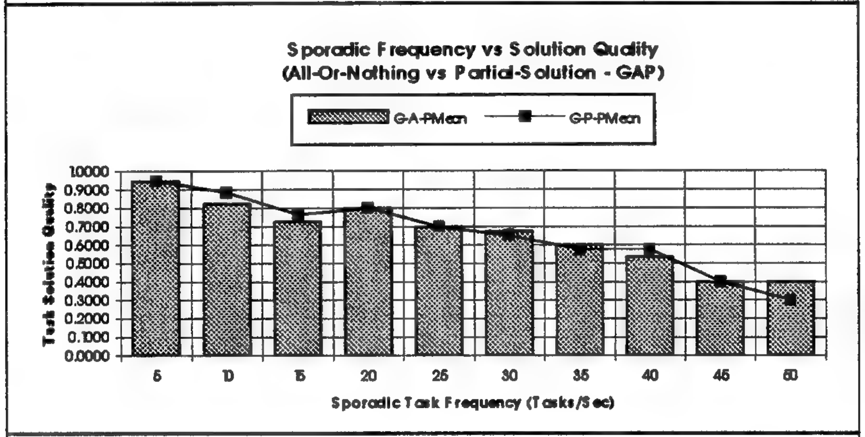


Figure D.1: Conservative-Duration Metric 1





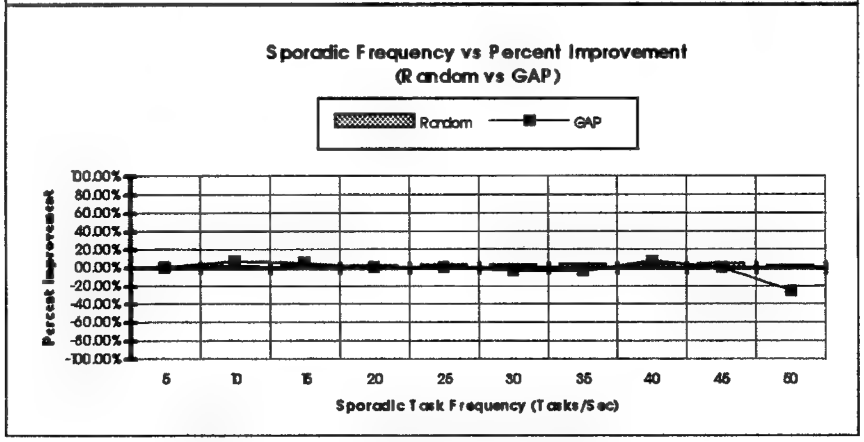


Figure D.2: Conservative-Frequency Metric 1

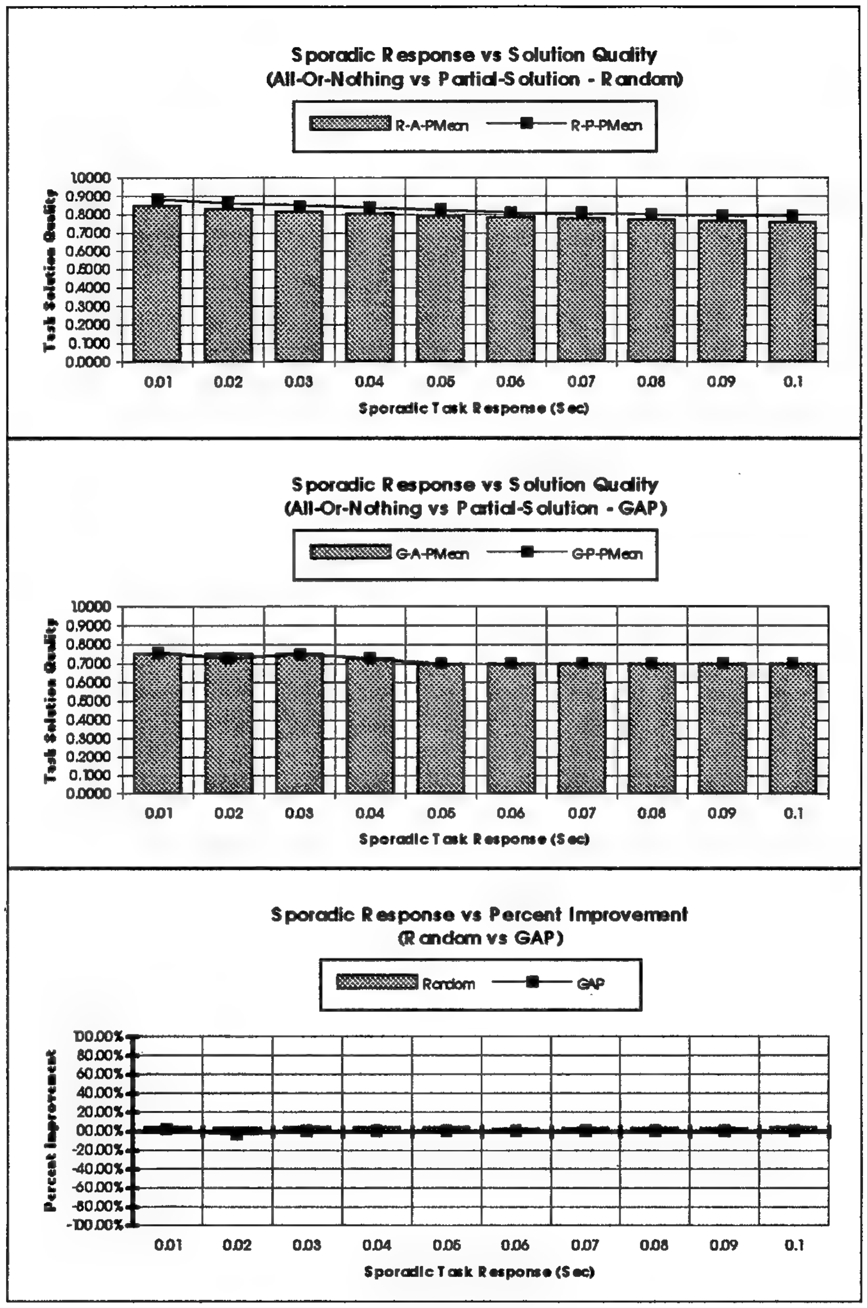
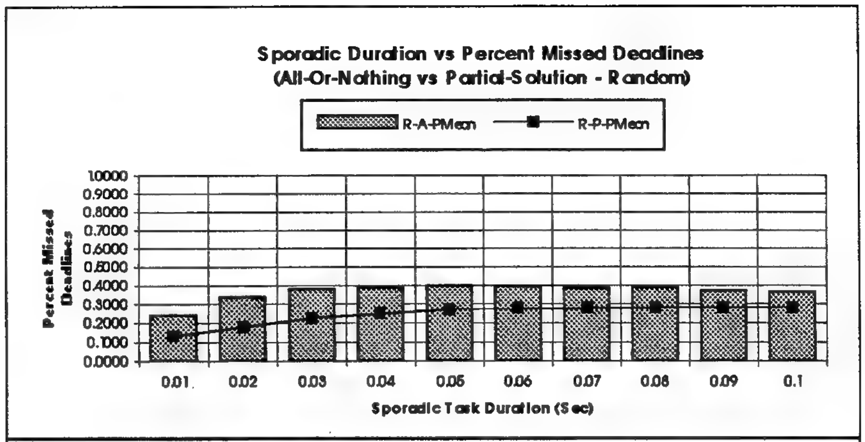
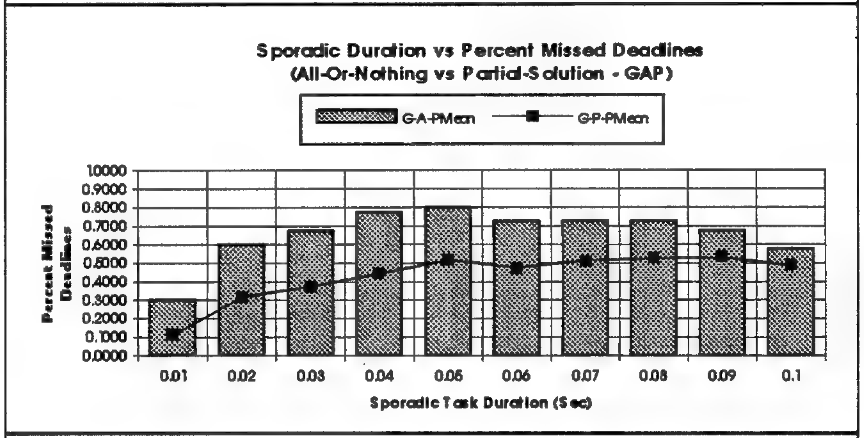


Figure D.3: Conservative-Response Metric 1

D.4 Conservative Results - Metric 2





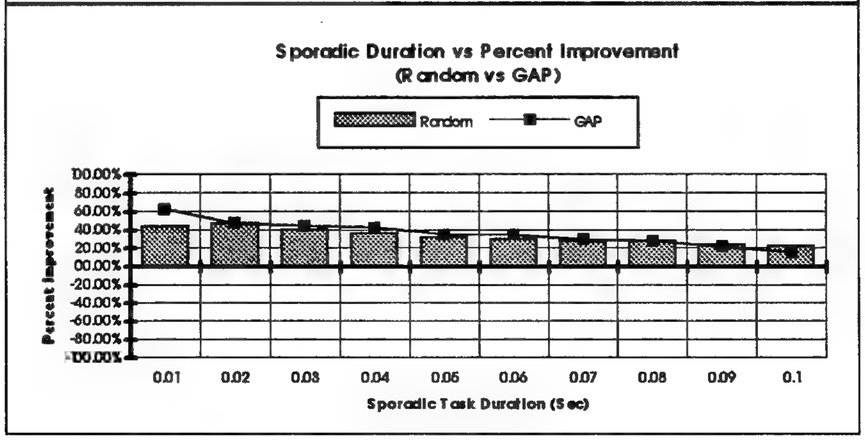


Figure D.4: Conservative-Duration Metric 2

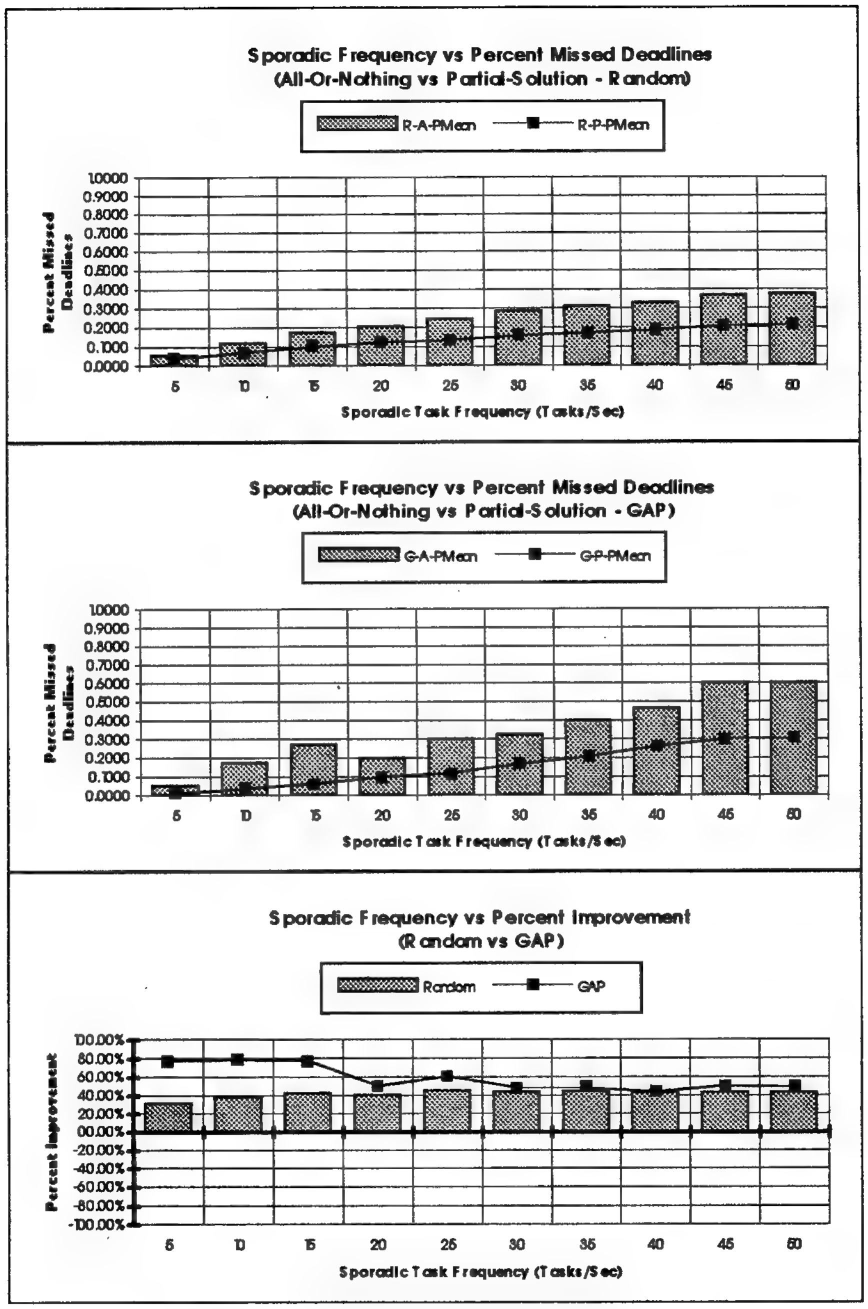


Figure D.5: Conservative-Frequency Metric 2

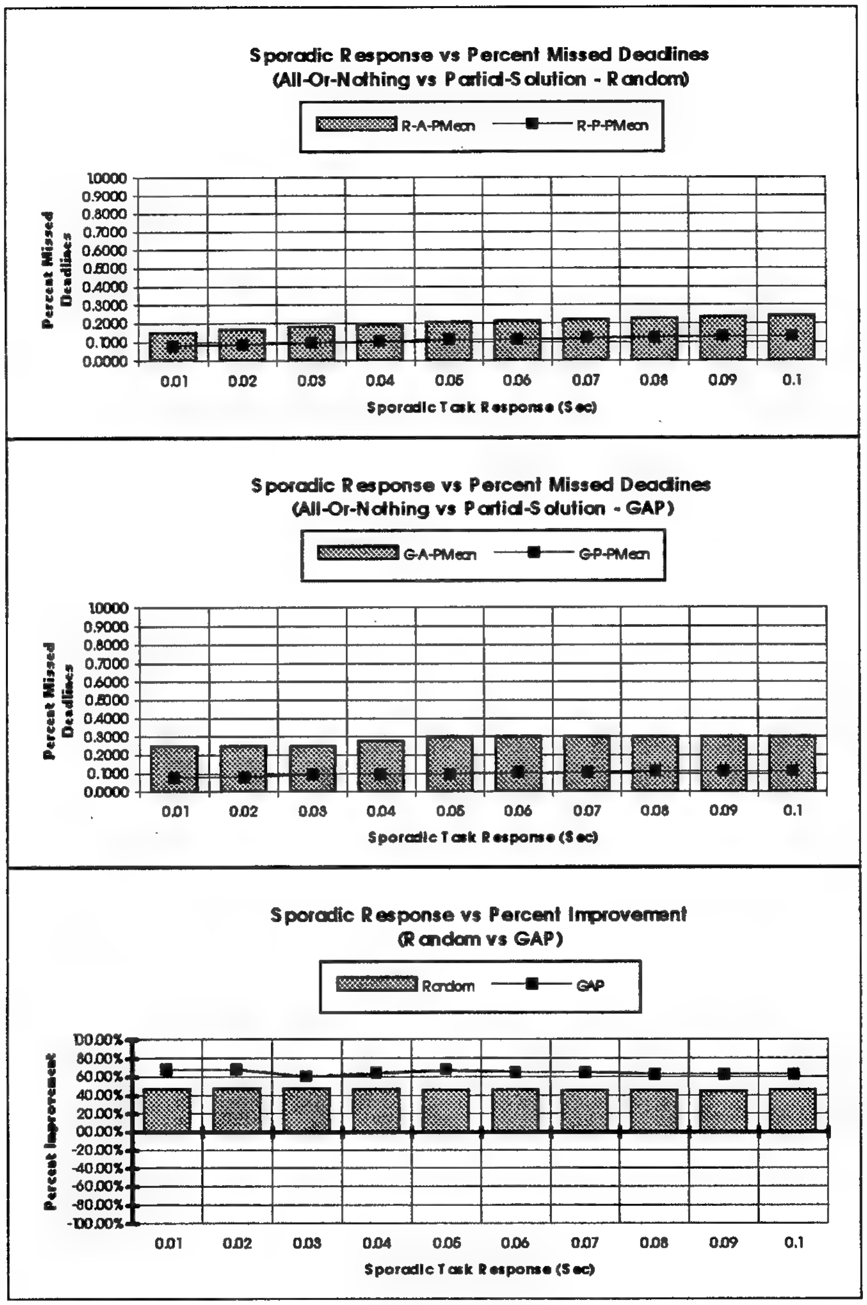


Figure D.6: Conservative-Response Metric 2

D.5 Linear Results - Metric 1

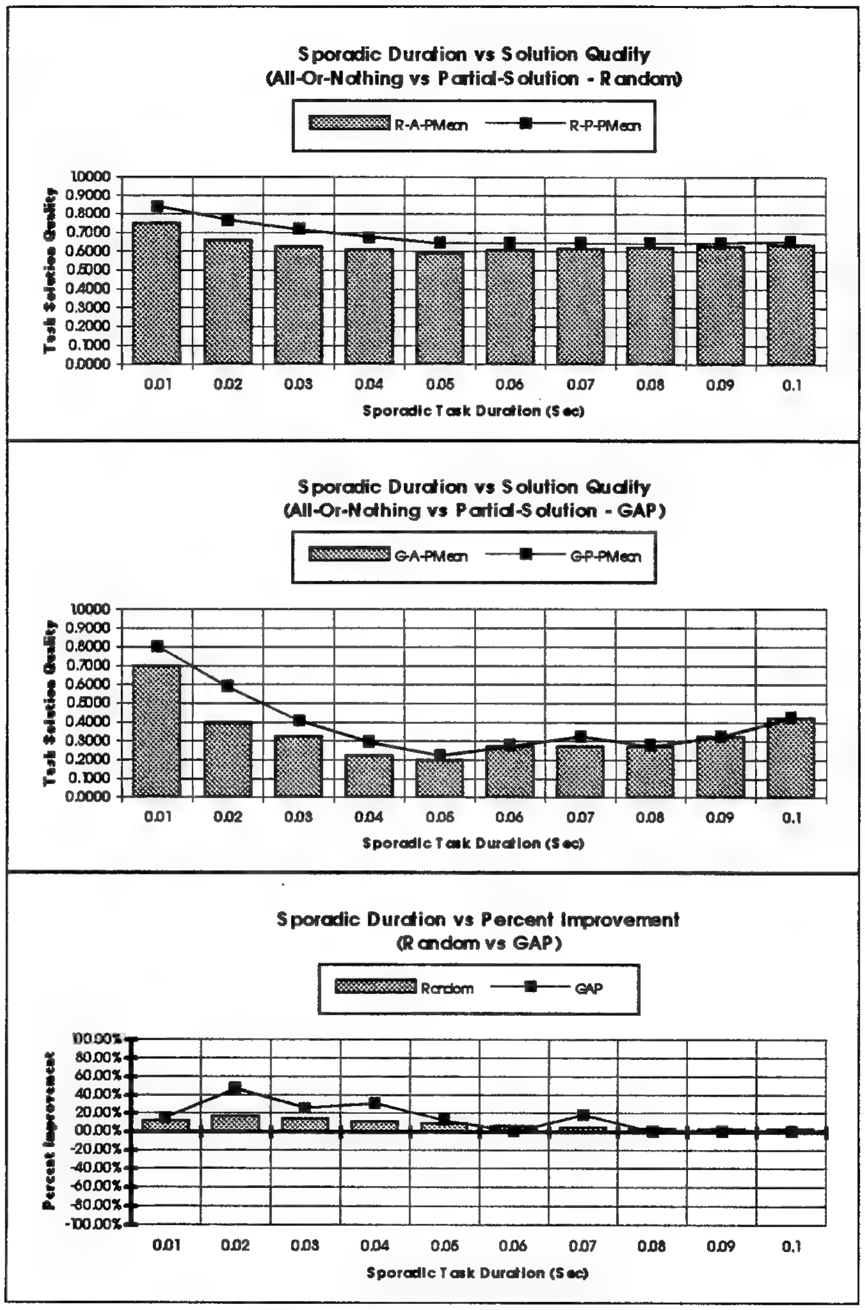


Figure D.7: Linear-Duration Metric 1

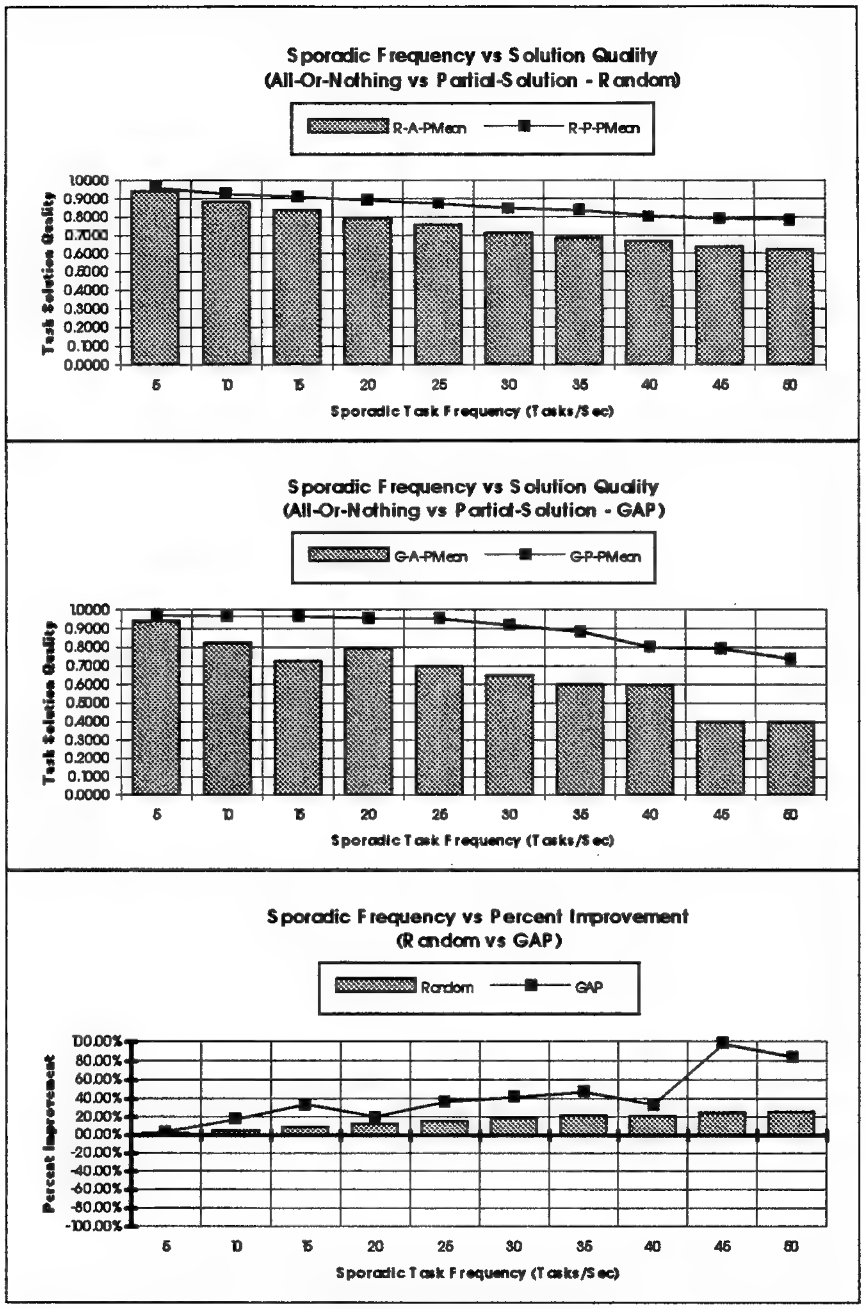


Figure D.8: Linear-Frequency Metric 1

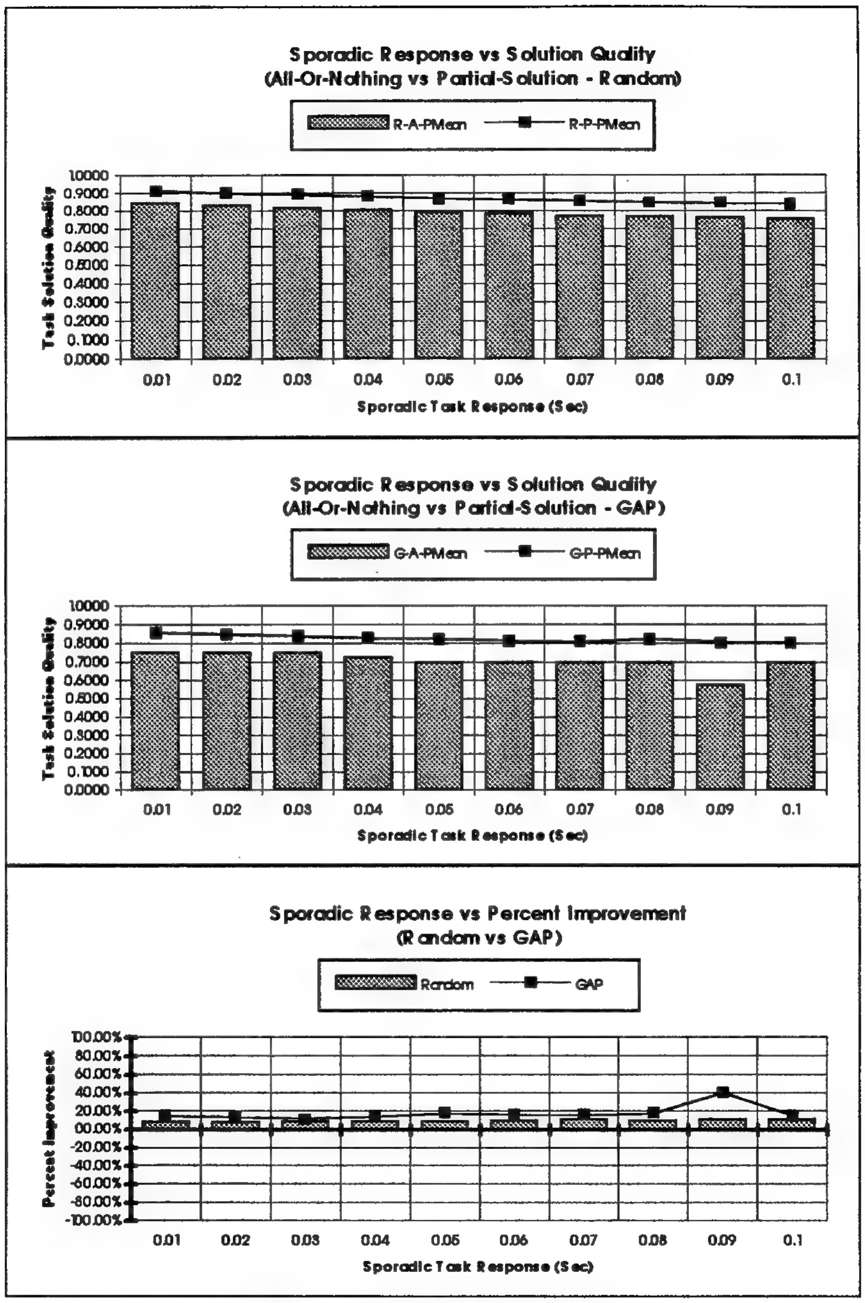


Figure D.9: Linear-Response Metric 1

D.6 Linear Results - Metric 2

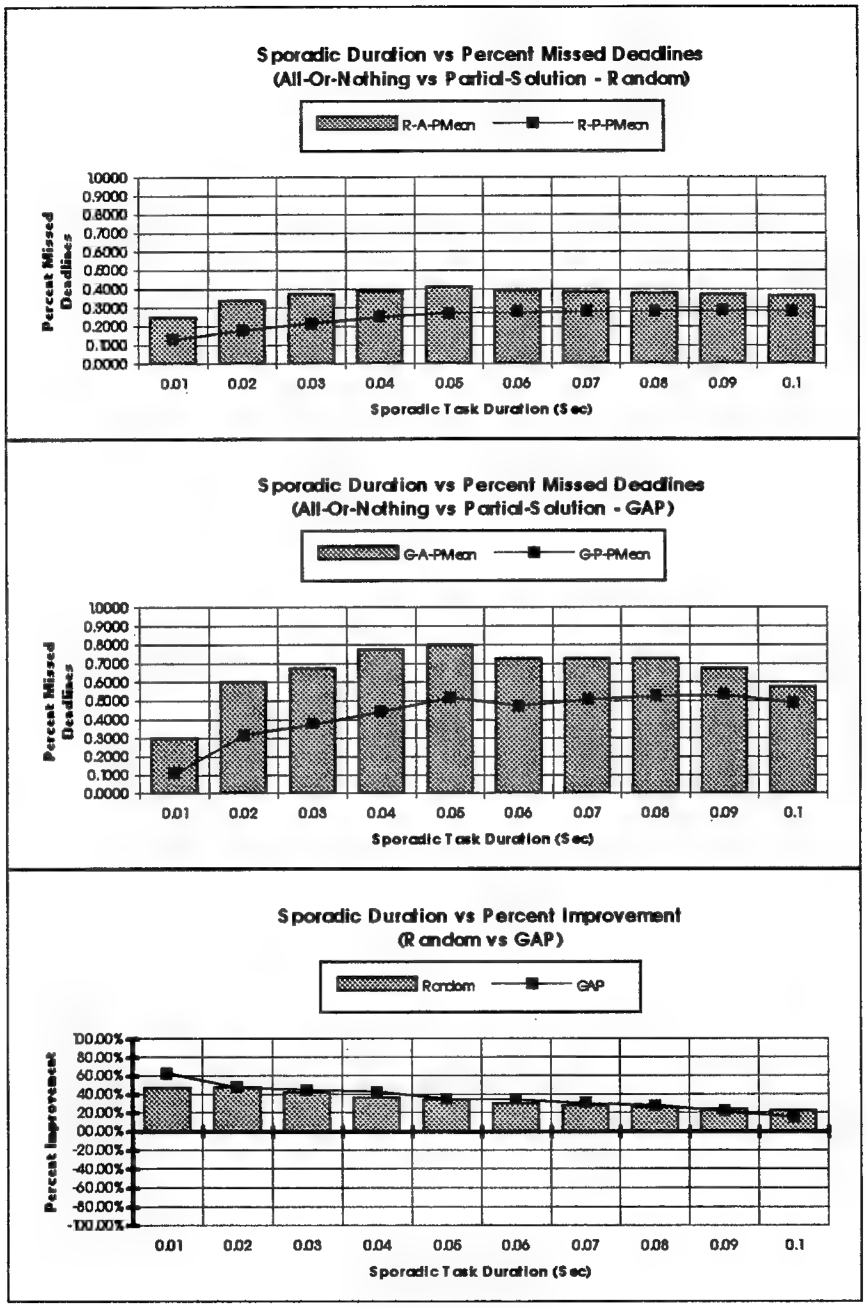


Figure D.10: Linear-Duration Metric 2

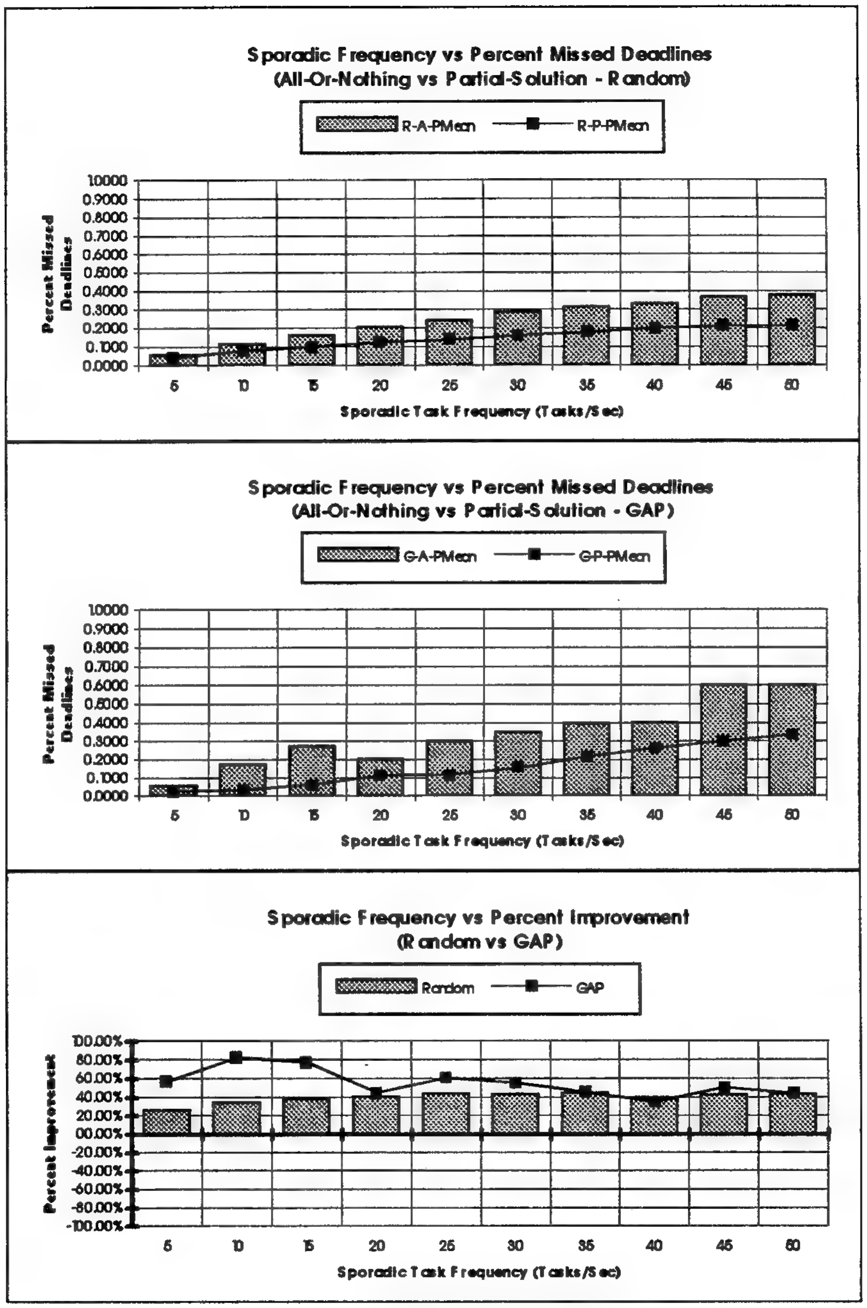


Figure D.11: Linear-Frequency Metric 2

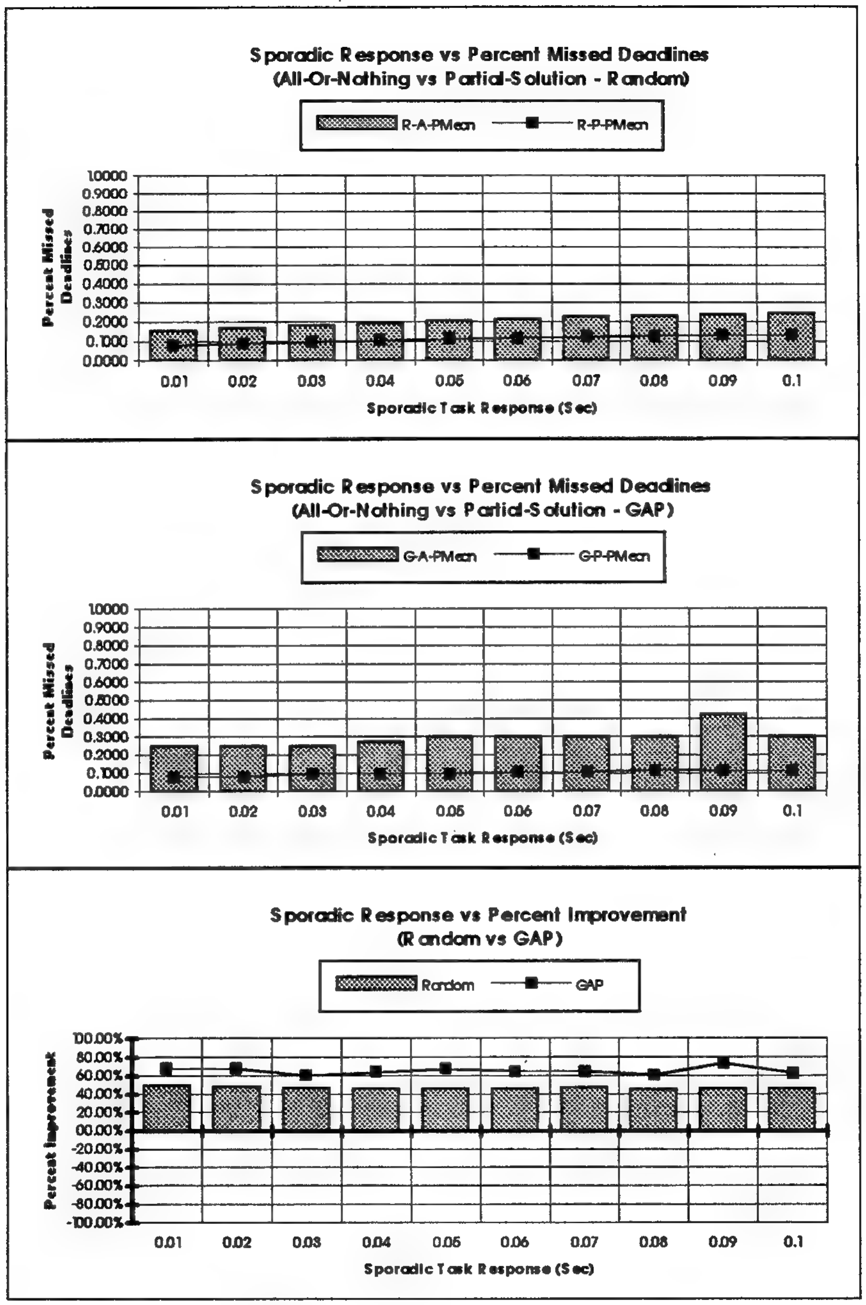


Figure D.12: Linear-Response Metric 2

D.7 Optimistic Results - Metric 1

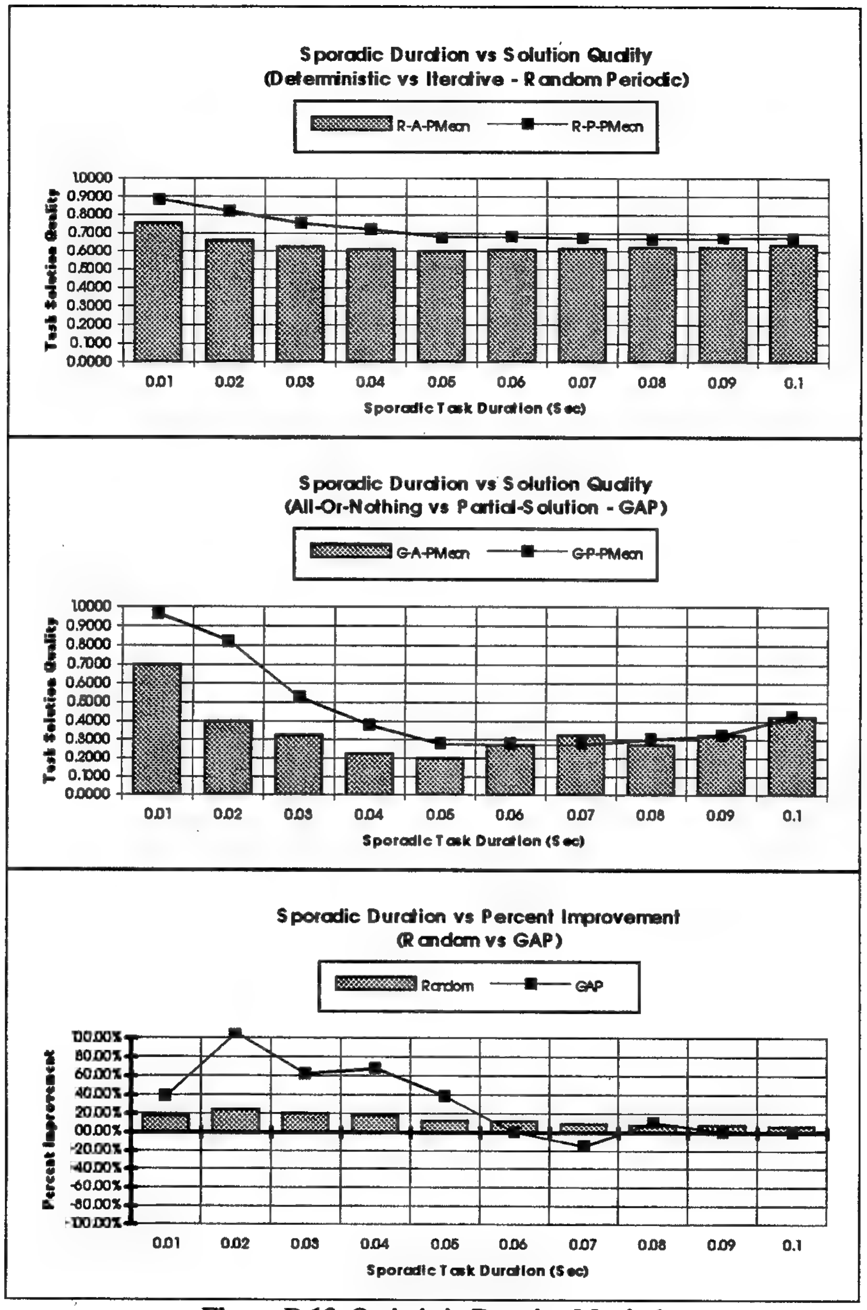
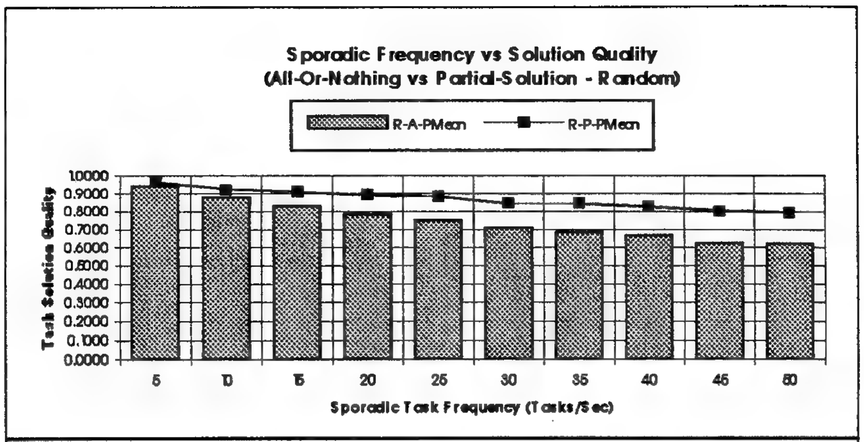
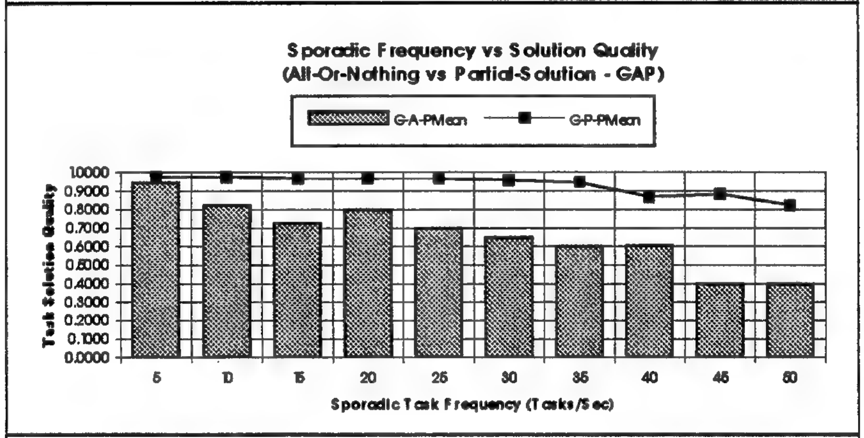


Figure D.13: Optimistic-Duration Metric 1





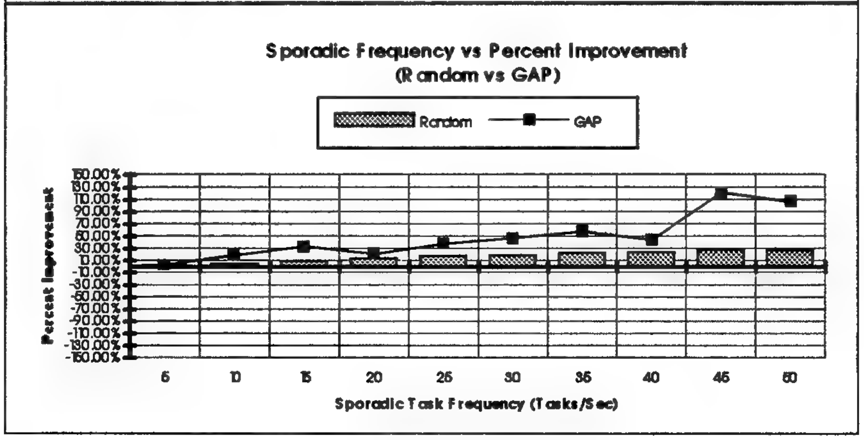


Figure D.14: Optimistic-Frequency Metric 1

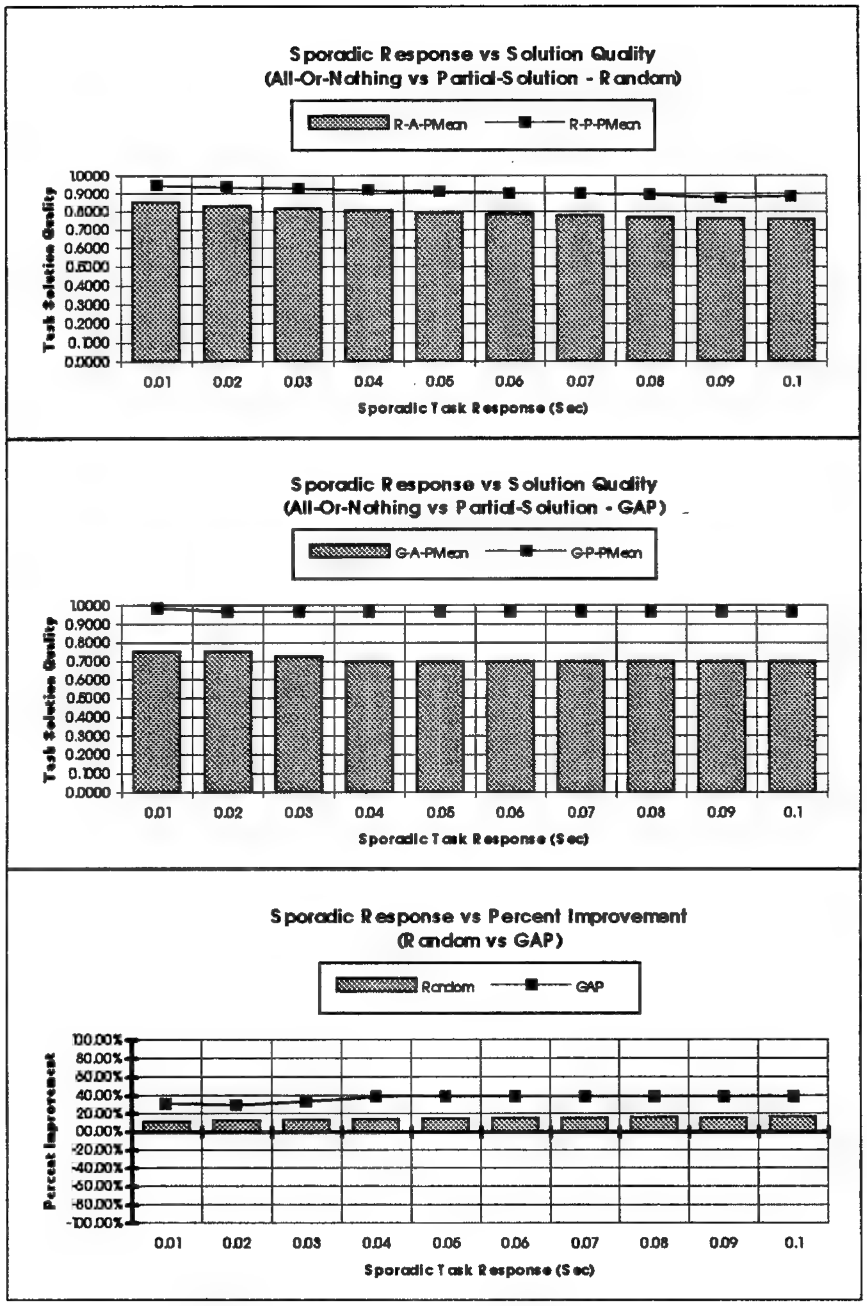
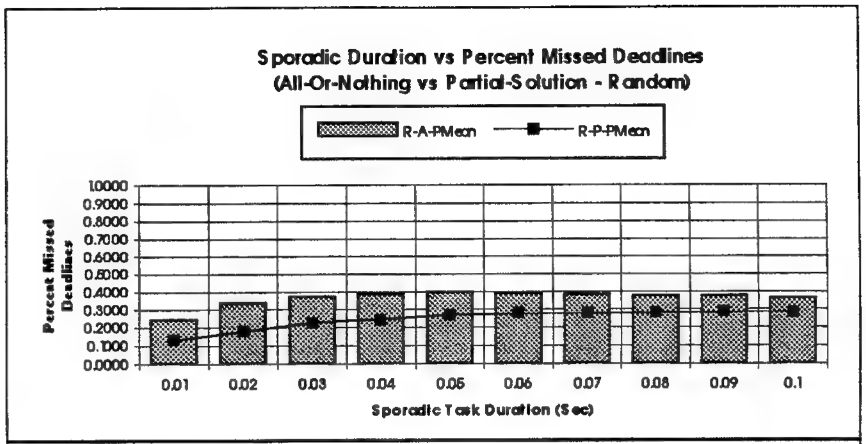
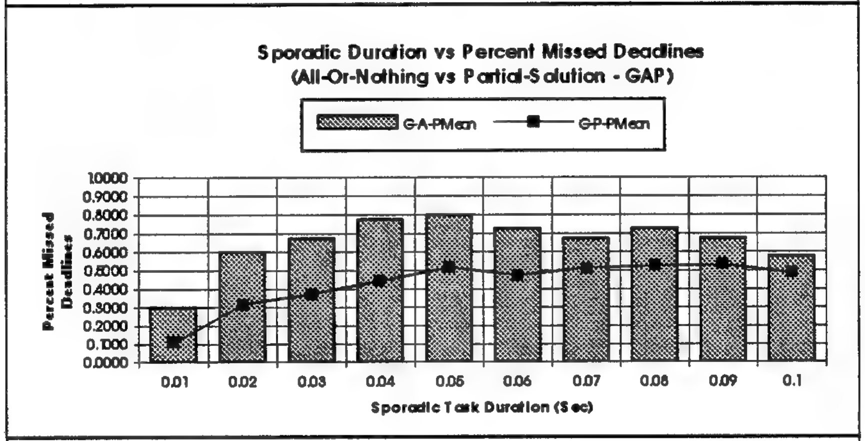


Figure D.15: Optimistic-Response Metric 1

D.8 Optimistic Results - Metric 2





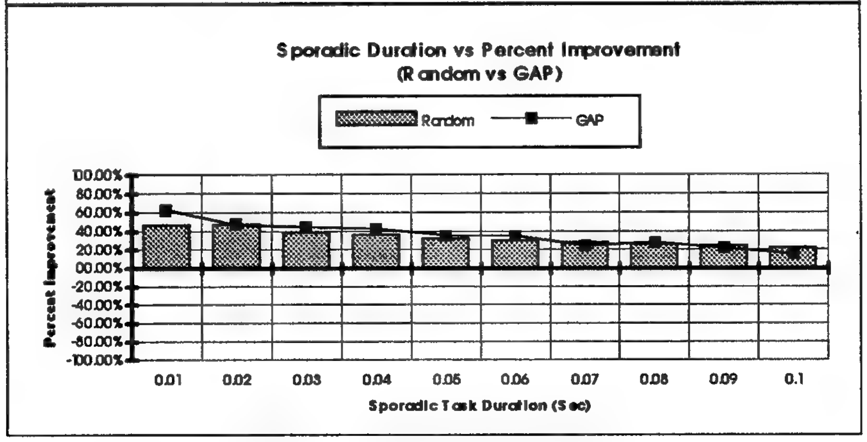


Figure D.16: Optimistic-Duration Metric 2

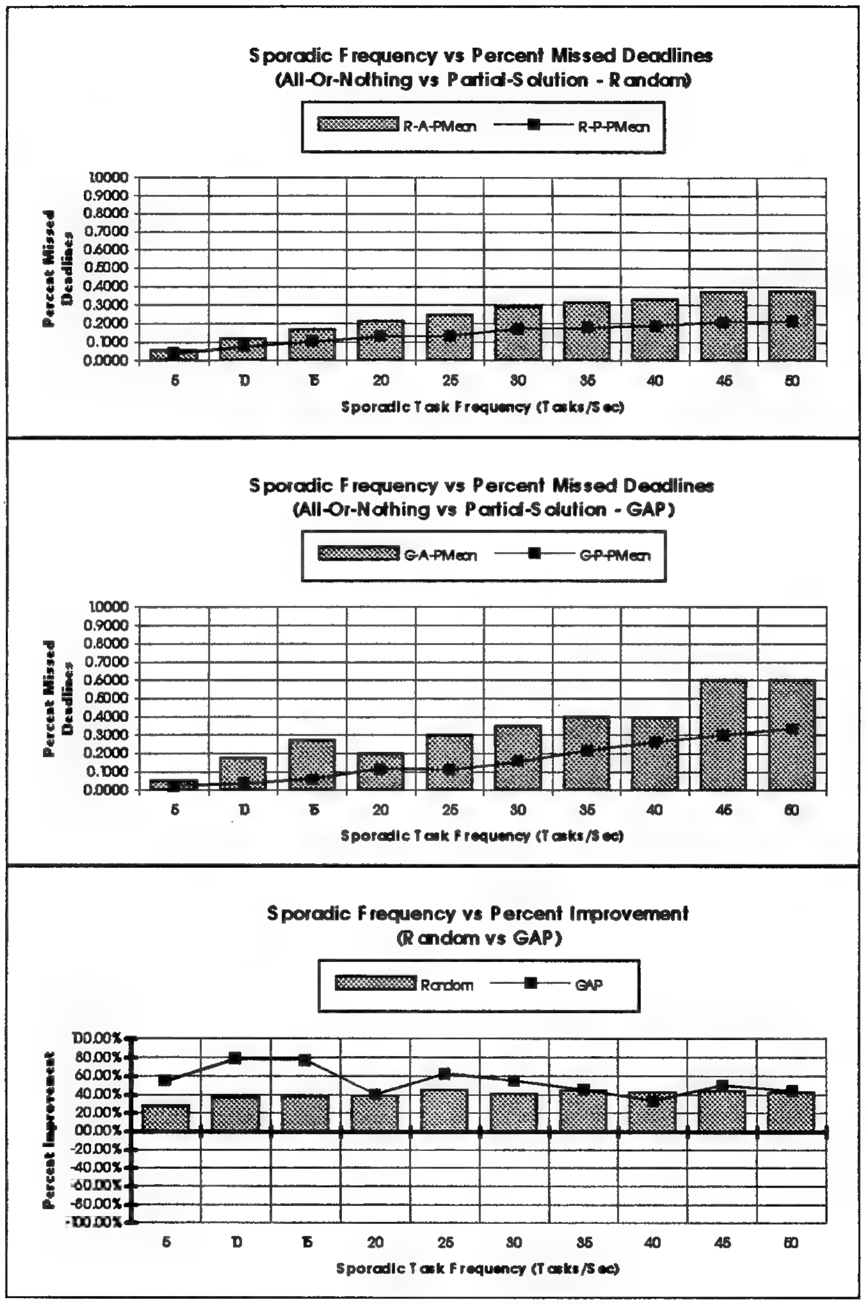


Figure D.17: Optimistic-Frequency Metric 2

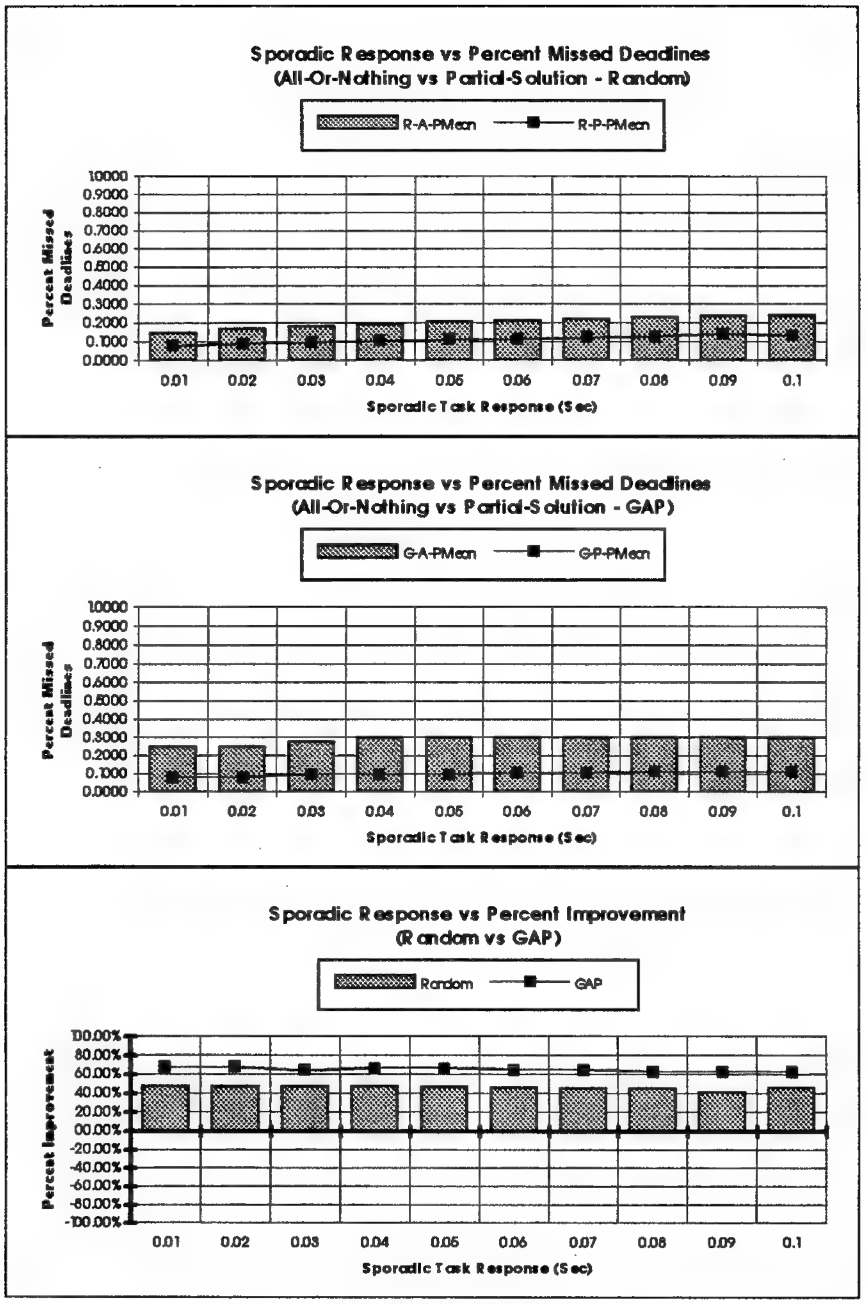


Figure D.18: Optimistic-Response Metric 2

Appendix E - Group Runtime Results

E.1 Description

This appendix presents group results from the runtime schedulability and robustness tests. Results are classified by result type (initial and extended), test type (duration, frequency, and response) and tasking requirement (random and GAP).

The runtime schedulability and robustness test was divided into two phases, scheduled and unscheduled. Results gathered during the scheduled phase of the test are referred to as the *initial results*. Results gathered during the unscheduled phase of the experiment are referred to as the *extended results*. The extended results were gathered in response to unexpected results during the scheduled phase of the runtime test. The process used for gathering these results was identical to the process used during the scheduled phase, with one exception. In the scheduled phase, all cyclic tasks were modeled as partial-solution tasks. In the unscheduled phase, only cyclic tasks with durations of 0.05 seconds or greater were modeled as partial-solution tasks, all others were modeled as all-or-nothing tasks. This modification was made to determine the impact of a small amount of domain-knowledge (task duration and executive resolution) on the performance of the partial-solution tasks.

Table E.1: Key

Metric 1	Average solution quality for cyclic tasks
Metric 2	Percent of missed deadlines for cyclic tasks
M-1-C	Metric 1, Conservative Curve
M-1-L	Metric 1, Linear Curve
M-1-O	Metric 1, Optimistic Curve
M-2-C	Metric 2, Conservative Curve
M-2-L	Metric 2, Linear Curve
M-2-O	Metric 2, Optimistic Curve

The purpose of the *group performance increase* graph is to allow visual comparison of the potential benefit that may be obtained using the Conservative, Linear, and Optimistic performance curves. In each case, these graphs suggest that best performance increase in solution quality, for partial-solution tasks, can be achieved using tasks that accomplish most of their quality early during execution.

A short description of terms used in this section is provided in Table E.1. A written summary of the results can be found in Chapter V.

E.2 Initial Duration Results

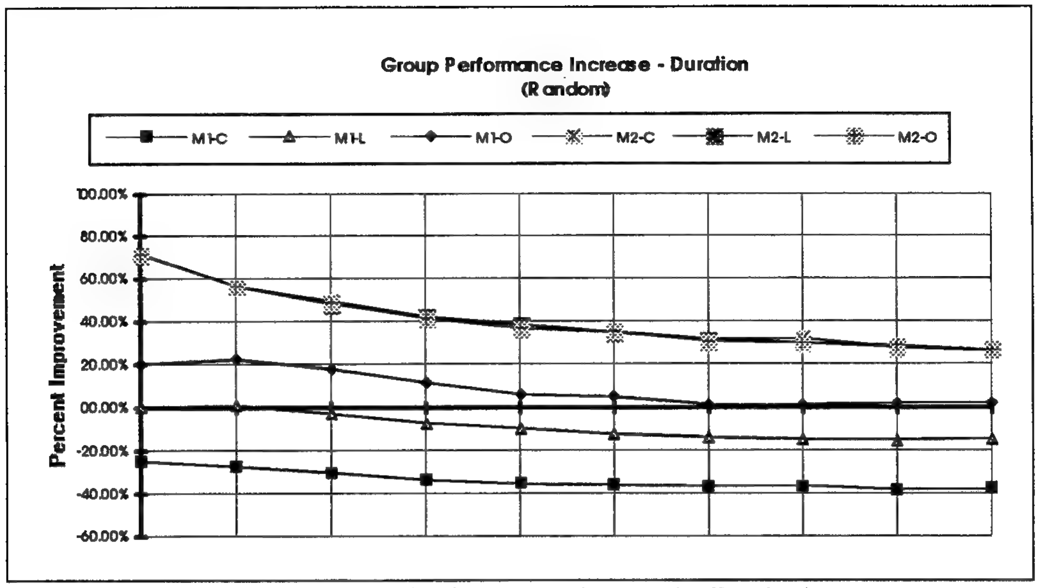


Figure E.1: Initial Group Duration Results (Random)

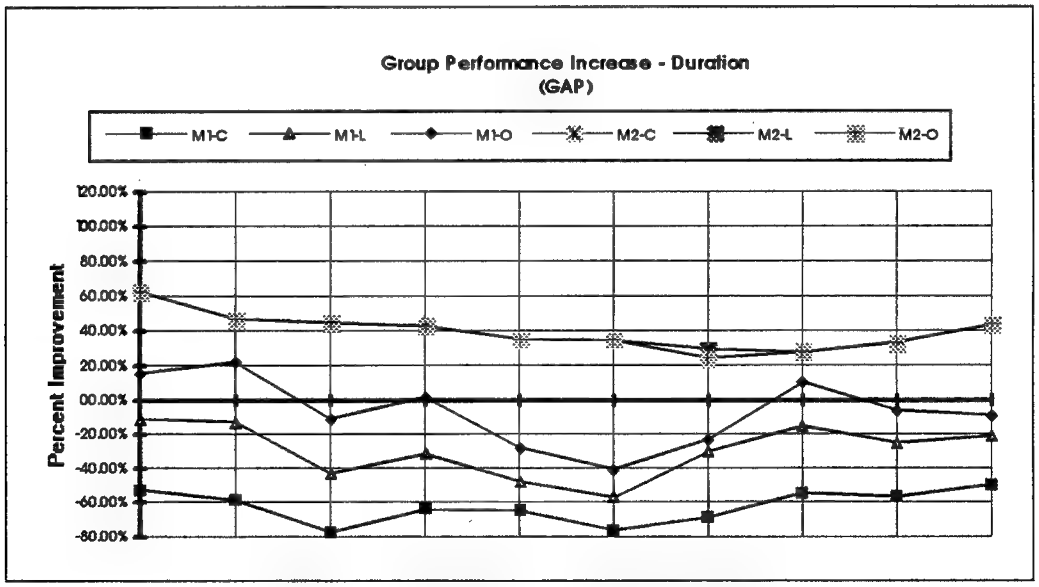


Figure E.2: Initial Group Duration Results (GAP)

E.3 Initial Frequency Results

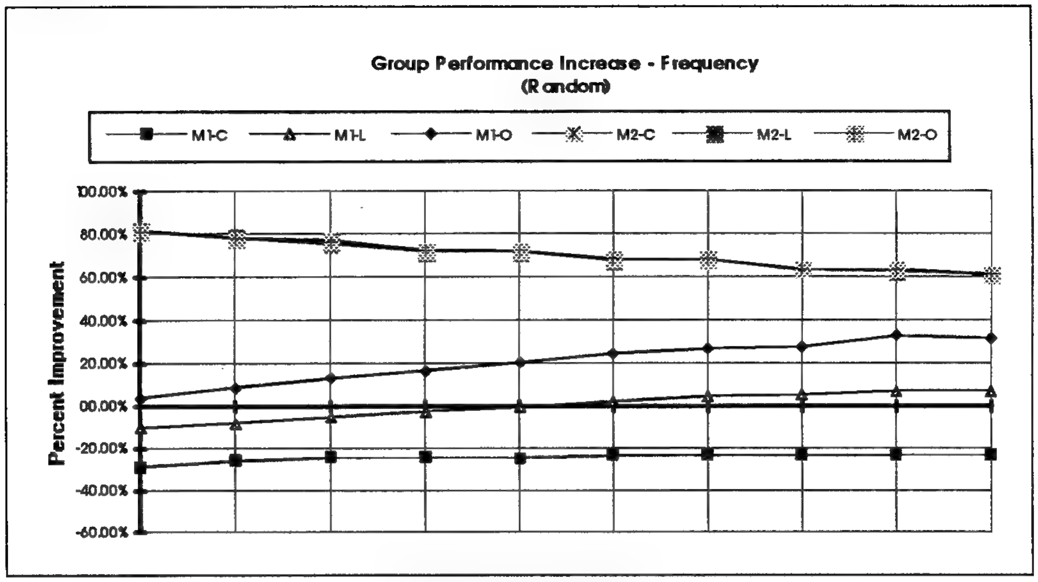


Figure E.3: Initial Group Frequency Results (Random)

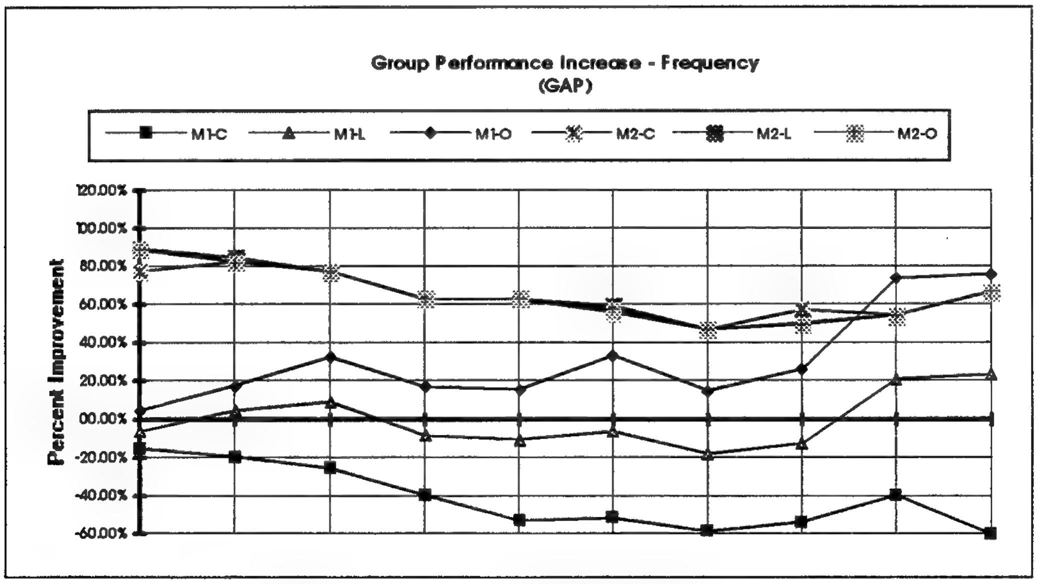


Figure E.4: Initial Group Frequency Results (GAP)

E.4 Initial Response Results

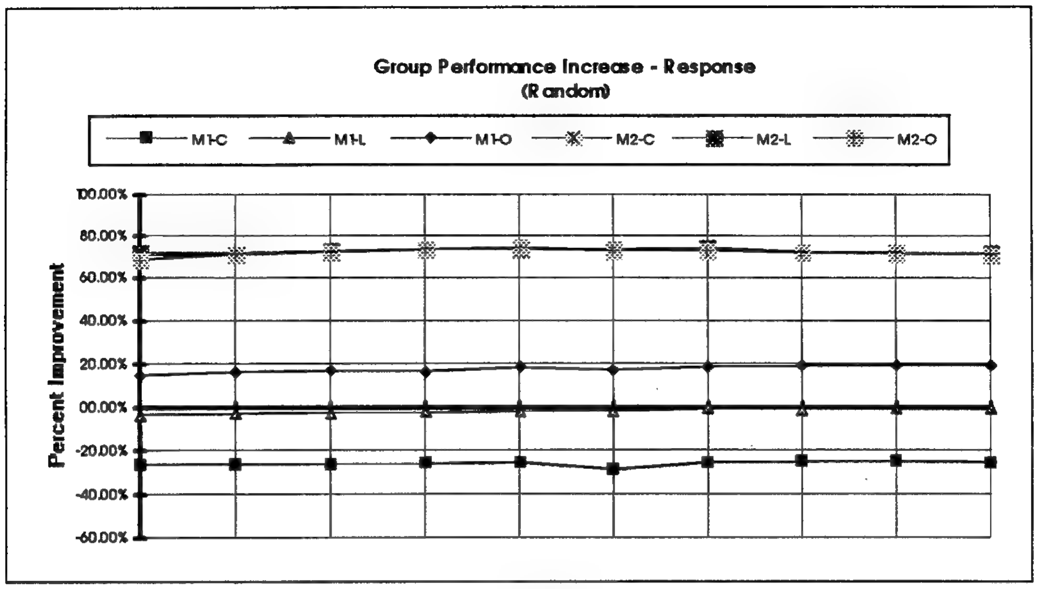


Figure E.5: Initial Group Response Results (Random)

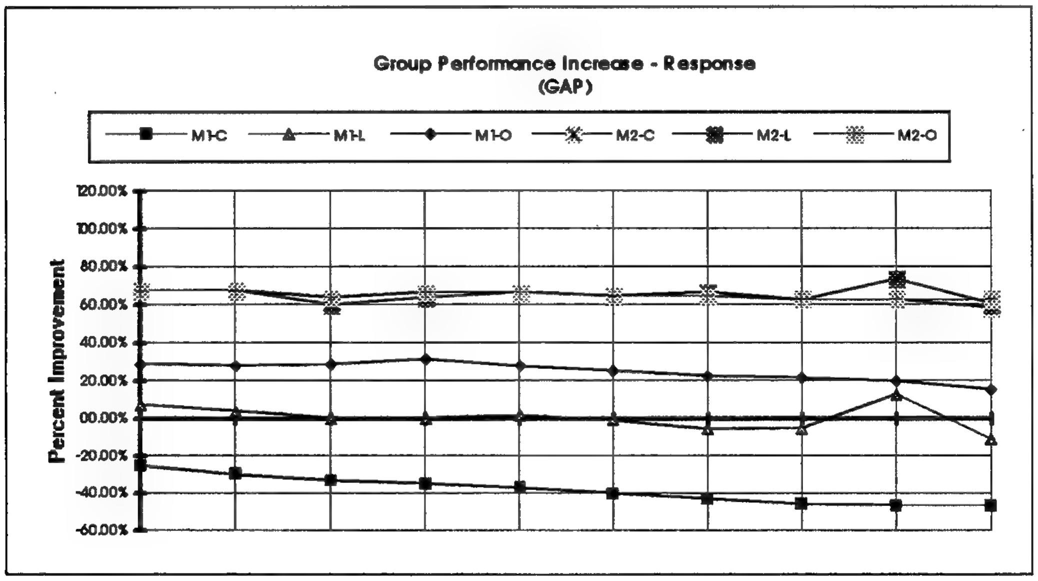


Figure E.6: Initial Group Response Results (GAP)

E.5 Extended Duration Results

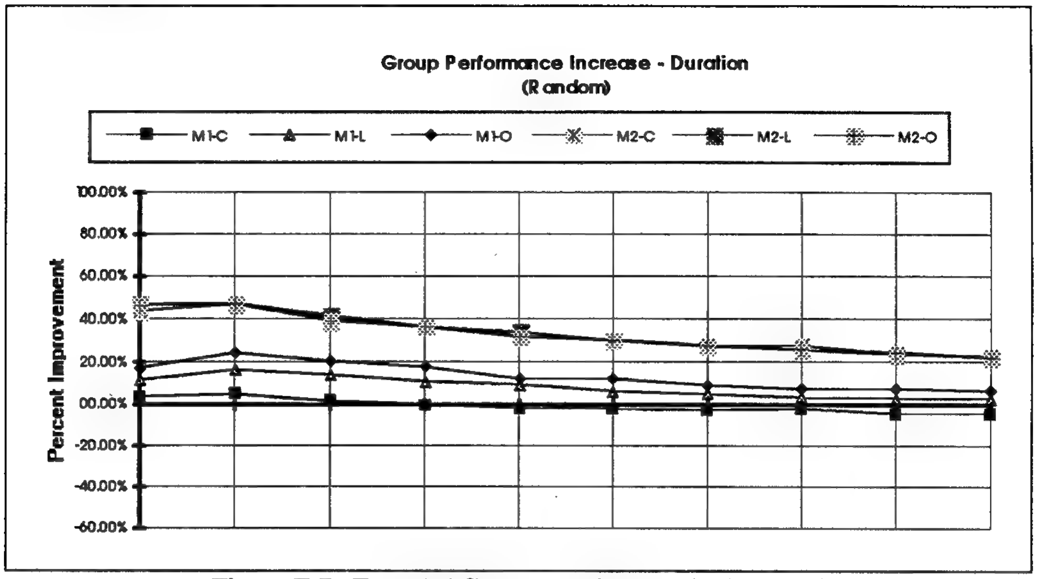


Figure E.7: Extended Group Duration Results (Random)

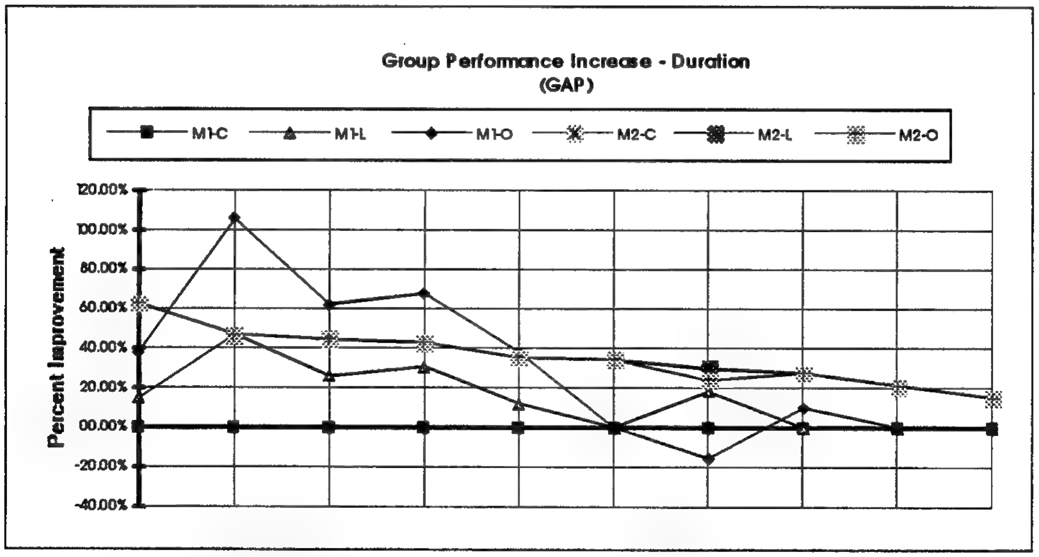


Figure E.8: Extended Group Duration Results (GAP)

E.6 Extended Frequency Results

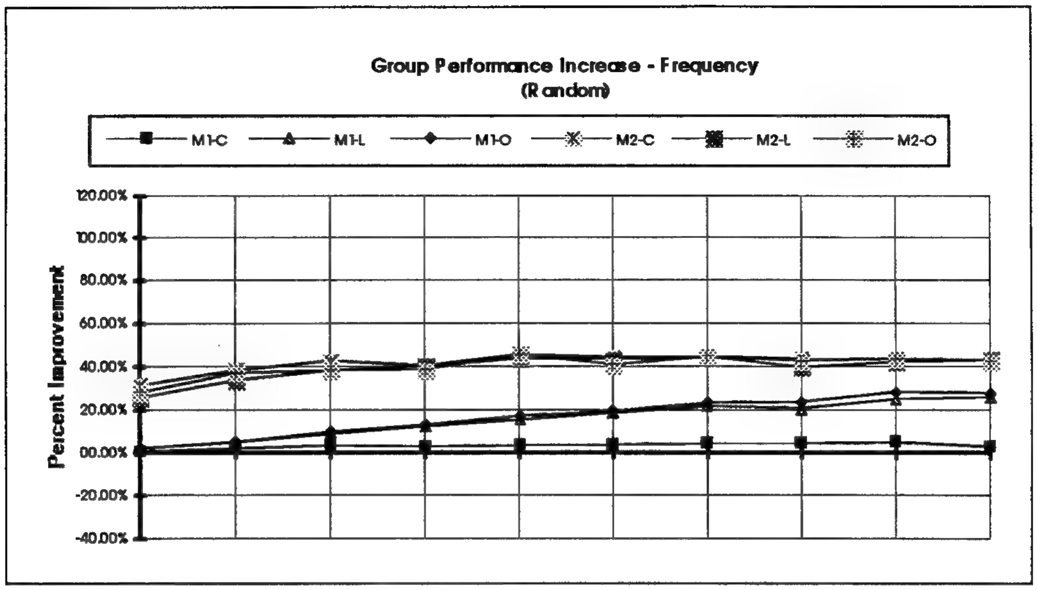


Figure E.9: Extended Group Frequency Results (Random)

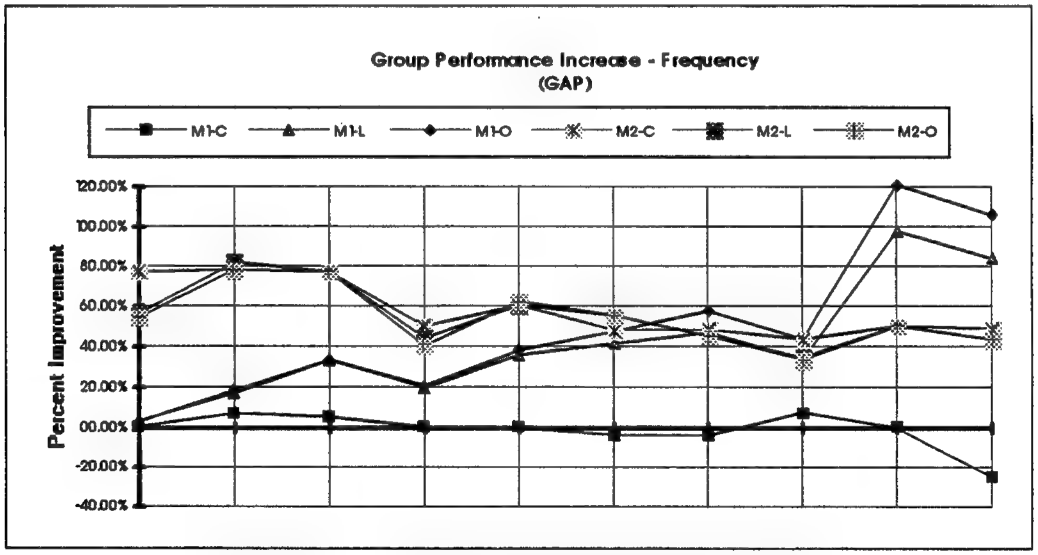


Figure E.10: Extended Group Frequency Results (GAP)

E.7 Extended Response Results

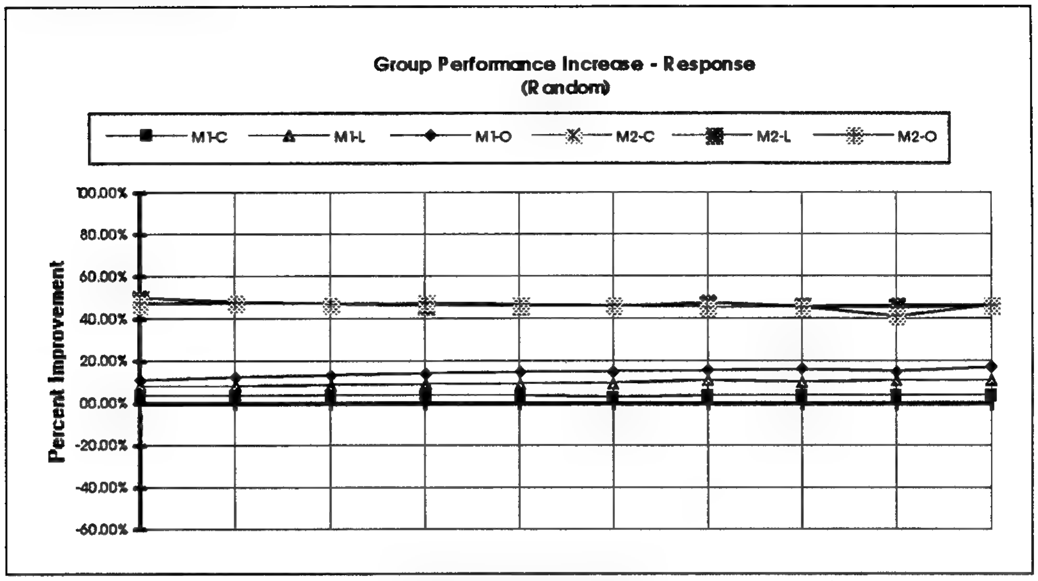


Figure E.11: Extended Group Response Results (Random)

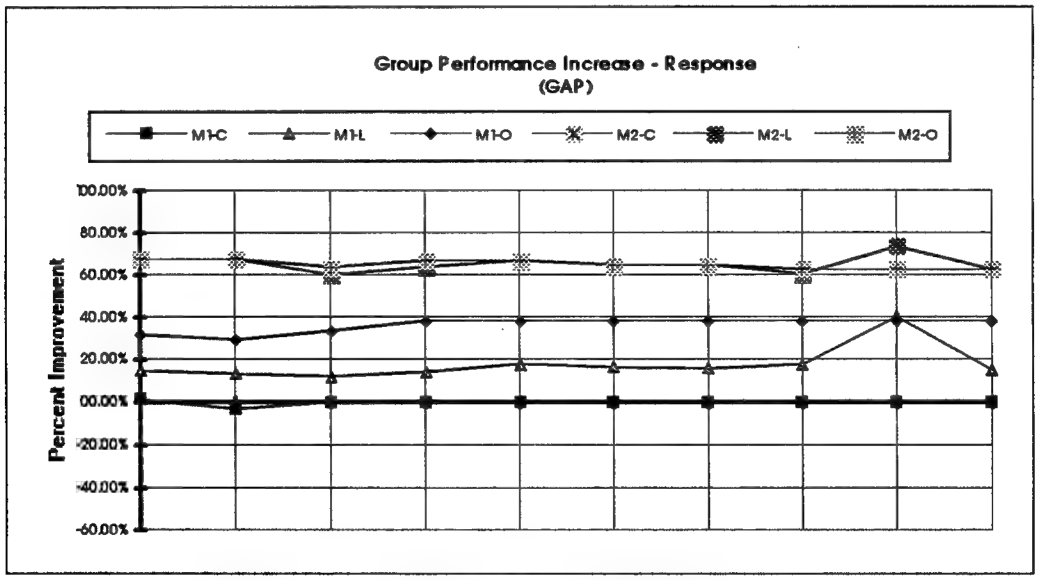


Figure E.12: Extended Group Response Results (GAP)

Appendix F - Selected Source Code

F.1 Description

This appendix contains selected source code for the support software used during the pre-runtime schedulability and maintainability demonstration and the runtime schedulability and robustness test. A written description of the software can be found in Chapter IV.

F.2 Runtime Executive

F.2.1 Timer - Package Specification

```
----------------------------------------------------------------------
-- @(#) /usr/eng/rcaley/Ada/Scheduler/Cyclic/SCCS/s.timer.ads 1.4 94/08/02
----------------------------------------------------------------------

package timer is

   timer_period_usec : constant integer := 10_000;  -- 0.010000 seconds
   alarms_per_second : constant integer := 100;
   seconds_per_alarm : constant float   := 0.01;
   alarms            : integer          := 0;

   procedure abort_task;
   procedure initialize_task;
   procedure set_timer;
   procedure stop_timer;

end timer;
----------------------------------------------------------------------
```

F.2.2 Timer - Package Body

```
----------------------------------------------------------------------
-- @(#) /rem3/94d/rcaley/Ada/Scheduler/Cyclic/SCCS/s.timer.adb 1.6 94/09/06
----------------------------------------------------------------------
with system;         use system;
with os_signal;      use os_signal;
with task_interface; use task_interface;
with resources;
with itimer;

with executive_task;

package body timer is

   task sigalrm_interrupt is
      entry initialize_task;
      entry SIGALRM;
         for SIGALRM use at address'ref(os_signal.SIGALRM);
      pragma priority(resources.priority_sigalrm);
   end sigalrm_interrupt;
   task body sigalrm_interrupt is
   begin
      accept initialize_task;

      loop
         accept SIGALRM do
            select
               executive_task.executive.resume_task;
            else
               missed(EXECUTIVE_ID) := missed(EXECUTIVE_ID) + 1;
            end select;
            attempted(EXECUTIVE_ID) := attempted(EXECUTIVE_ID) + 1;
         end SIGALRM;
         alarms := alarms + 1;
         set_timer;
      end loop;
   end sigalrm_interrupt;

   procedure initialize_task is
   begin
      sigalrm_interrupt.initialize_task;
   end initialize_task;

   procedure set_timer is
      newitimer   : itimer.itimerval_rec;
      olditimer   : itimer.itimerval_rec;
      status      : integer;
   begin
      itimer.fillitimer(newitimer, 0, 0, 0, timer_period_usec);
      status := itimer.setitimer(itimer.ITIMER_REAL,
         itimer.to_itimerval_ptr(newitimer'address),
         itimer.to_itimerval_ptr(olditimer'address));
   end set_timer;

   procedure stop_timer is
     newitimer   : itimer.itimerval_rec;
     olditimer   : itimer.itimerval_rec;
     status      : integer;
   begin
     itimer.fillitimer(newitimer, 0, 0, 0, 0);
     status := itimer.setitimer(itimer.ITIMER_REAL,
         itimer.to_itimerval_ptr(newitimer'address),
         itimer.to_itimerval_ptr(olditimer'address));
   end stop_timer;

   procedure abort_task is
   begin
     stop_timer;
     abort sigalrm_interrupt;
   end abort_task;

end timer;
-------------------------------------------------------------------------
```

F.2.3 Executive - Package Specification

```
-------------------------------------------------------------------------
-- @(#) /usr/eng/rcaley/Ada/Scheduler/Cyclic/SCCS/s.executive.ads 94/08/09
-------------------------------------------------------------------------
with resources;

package executive_task is
   task executive is
      entry initialize_task;
      entry resume_task;
      pragma priority(resources.priority_executive);
   end executive;

   procedure initialize_task;
   procedure resume_task;
end executive_task;
-------------------------------------------------------------------------
```

F.2.4 Executive - Package Body

```
----------------------------------------------------------------------
--@(#)/rem3/94d/rcaley/Ada/Scheduler/Cyclic/SCCS/s.executive.adb 94/09/06
----------------------------------------------------------------------
with Timer;
with system;         use system;
with unix;           use unix;
with os_time;        use os_time;
with v_xtasking;     use v_xtasking;
with text_io;        use text_io;
with calendar;       use calendar;
with task_schedule;  use task_schedule;
with task_interface; use task_interface;
with xcalendar;
with resources;
with sporadic_generator;
with sporadic_scheduler;

package body executive_task is

   procedure complete_task is
   begin
     task_interface.update_task(DISPLAY_ID, TASK_ABORT);
     unix.sys_exit;
   end complete_task;

   procedure initialize_task is
   begin
      executive.initialize_task;
   end initialize_task;

   procedure resume_task is
   begin
      executive.resume_task;
   end resume_task;

   task body executive is
   begin
      accept initialize_task;
      Timer.set_timer;

      for major_loop in 1..task_interface.test_duration loop
         schedule_position := 1;
         while schedule_position <= schedule_length loop

            accept resume_task;

            sporadic_generator.generate_activation_list(sporadic_frequency);
            sporadic_scheduler.queue_sporadic_tasks;
            sporadic_scheduler.resume_interrupted_task;

            -- Process Task Actions for Current Time Cycle
            if (schedule(schedule_position).num > 0) then
               for action_count in 1..schedule(schedule_position).num loop
                  task_interface.update_task(
                  schedule(schedule_position).actions(action_count).id,
                   schedule(schedule_position).actions(action_count).action);
               end loop;
            end if;

            sporadic_scheduler.schedule_sporadic_tasks;

            time_used(running_task) := time_used(running_task) +
               timer.seconds_per_alarm;
            schedule_position := schedule_position + 1;
         end loop; -- while
      end loop; -- for

      complete_task;
   end executive;
end executive_task;
----------------------------------------------------------------------
```

F.2.5 Generic_Task - Package Specification

```
---------------------------------------------------------------------------
--@(#)/rem3/94d/rcaley/Ada/Scheduler/Cyclic/SCCS/s.generic.ads 94/09/23
---------------------------------------------------------------------------
with task_interface;
with system;

generic
   id : task_interface.task_id;

package generic_task is

   procedure task_abort;
   procedure task_initialize;
   procedure task_start;
   procedure task_stop;
   procedure task_resume;
   procedure task_suspend;
   procedure task_stopwork;

   function get_task_id return system.task_id;
   function task_completed return boolean;
end generic_task;
---------------------------------------------------------------------------
```

F.2.6 Generic_Task - Package Body

```
---------------------------------------------------------------------------
--@(#)/rem3/94d/rcaley/Ada/Scheduler/Cyclic/SCCS/s.generic.adb 94/09/23
---------------------------------------------------------------------------
with timer;
with task_interface; use task_interface;
with text_io;        use text_io;
with system;         use system;
with io_package;     use io_package;
with v_xtasking;
with v_i_tasks;
with resources;

package body generic_task is

   task_id      : system.task_id;
   completed    : boolean := TRUE;
   stopwork     : boolean := FALSE;
   suspended    : boolean := FALSE;
   reset        : boolean := TRUE;
   lcount       : integer := 1000;
   conservative : array (1..20) of float :=
      (0.0000, 0.0000, 0.0001, 0.0002, 0.0003,
       0.0006, 0.0013, 0.0026, 0.0051, 0.0102,
       0.0205, 0.0409, 0.0817, 0.1624, 0.3164,
       0.5753, 0.8645, 0.9893, 0.9999, 1.0000);
   linear       : array (1..20) of float :=
      (0.0500, 0.1000, 0.1500, 0.2000, 0.2500,
       0.3000, 0.3500, 0.4000, 0.4500, 0.5000,
       0.5500, 0.6000, 0.6500, 0.7000, 0.7500,
       0.8000, 0.8500, 0.9000, 0.9500, 1.0000);
   optimistic   : array (1..20) of float :=
      (0.0000, 0.0000, 0.0001, 0.0105, 0.1353,
       0.4247, 0.6836, 0.8376, 0.9183, 0.9591,
       0.9795, 0.9898, 0.9949, 0.9974, 0.9987,
       0.9994, 0.9997, 0.9998, 0.9999, 1.0000);
```

```
task doit is
   entry initialize_task;
   entry start_next_cycle;
   pragma priority(resources.priority_worker);
end doit;

task body doit is
   temp      : integer := 1;
   iteration : integer := 0;
begin
   task_id := v_i_tasks.get_current_task;
   accept initialize_task;


   loop
      accept start_next_cycle do
         completed    := FALSE;
         stopwork     := FALSE;
         reset        := TRUE;
         iteration    := 0;
         loops(id)    := 0;
         quality(id) := 0.0;
      end start_next_cycle;

      while stopwork = FALSE loop
         if reset then
            reset := FALSE;
            iteration   := 0;
            loops(id)   := 0;
            quality(id) := 0.0;
         end if;

         for count in 1..lcount  loop
            if stopwork or reset then
               exit;
            end if;
            temp := (temp + 1) mod 101;
            loops(id) := loops(id) + 1;
         end loop;

         iteration    := iteration + 1;
         if ttype(id) = ITERATIVE_TASK then
            quality(id) :=  conservative(iteration);
         end if;
         exit when iteration = 20;
      end loop;
      if (ttype(id) = ITERATIVE_TASK) then
         if (iteration /= 0) then
            quality(id) :=  conservative(iteration);
         end if;
      else
         if (iteration = 20) then
            quality(id) := 1.0;
         end if;
      end if;

      completed := TRUE;

  end loop;
end doit;

procedure task_initialize is
begin
  doit.initialize_task;
end task_initialize;

procedure task_abort is
begin
   abort doit;
   if suspended then
      v_xtasking.resume_task(task_id);
      suspended := FALSE;
   end if;
end task_abort;
```

```
procedure task_start is
   fcount : float;
begin
   fcount      := durations(id) * float(loops_for_second) / 20.0;
   lcount      := integer(fcount);
   reset       := TRUE;

   loops(id)   := 0;
   quality(id) := 0.0;

   if suspended then
      v_xtasking.resume_task(task_id);
      suspended := FALSE;
   end if;

   if completed then
      doit.start_next_cycle;
   end if;

   status(id)    := task_interface.RUNNING;
   attempted(id) := attempted(id) + 1;
end task_start;

procedure task_stop is
begin
   if completed then
      status(id)   := task_interface.COMPLETED;
   else
      status(id) := task_interface.SUSPENDED;
      v_xtasking.suspend_task(task_id);
      suspended := TRUE;
      missed(id) := missed(id) + 1;
   end if;

   tquality(id) := tquality(id) + quality(id);
   tloops(id)   := tloops(id) + loops(id);
end task_stop;

procedure task_resume is
begin
   if suspended then
      v_xtasking.resume_task(task_id);
      suspended := FALSE;
   end if;

   if completed then
      status(id) := task_interface.COMPLETED;
   else
      status(id) := task_interface.RUNNING;
   end if;
end task_resume;

procedure task_suspend is
begin
   if completed then
      status(id) := task_interface.COMPLETED;
   else
      status(id) := task_interface.SUSPENDED;
      v_xtasking.suspend_task(task_id);
      suspended := TRUE;
   end if;
end task_suspend;
```

```
   procedure task_stopwork is
   begin
      if (ttype(id) = ITERATIVE_TASK) and (not completed) then
         stopwork := TRUE;
      end if;
   end task_stopwork;

   function get_task_id return system.task_id is
   begin
      return task_id;
   end get_task_id;

   function task_completed return boolean is
   begin
     return completed;
   end task_completed;

end generic_task;
-------------------------------------------------------------------------
```

F.3 Pre-Runtime Scheduler

F.3.1 schedule.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; File:        schedule.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Data Structures:
;;   system     (processor processor processor ...)
;;   processor  (process process process ...)
;;   process    (task duration processor)
;;   task       (name duration period importance priority
;;               bodyin bodyout specin specout)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Running the Program:
;;
;; Print Sched - (print-schedule num_processors task task ... task)
;;    Example:  (print-schedule 1 t1 t2 t3)
;;
;; Translate Sched - (translate-schedule num_processors task task ... task)
;;    Example:  (translate-schedule 1 t1 t2 t3)
;;
;; Translation Output:  Translations are currently stored in the Source
;;                      directory.
;;
;; Template Files:  Ada templates are currently stored in the Template
;;                  directory.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Documentation:  For additional information on rate monotonic scheduling
;;                 read:
;;
;;    Levi, Shem-Tov and Ashok K. Agrawala, "Real-Time System Design,"
;;        McGraw-Hill Publishing Company, 1990.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "USER")
(provide "schedule")

(require "task")
(require "print")
(require "translate")

;------------------------------------------------------------------------
(defconstant MIN-PRIORITY 39)
(defconstant MAX-PRIORITY 30)
;------------------------------------------------------------------------

(defun print-schedule(num-processors &rest task-list)
   (print-system (schedule-system num-processors task-list)))

(defun translate-schedule(num-processors &rest task-list)
   (translate-system (schedule-system num-processors task-list)))
```

```
;---------------------------------------------------------------------------
; Scheduling Algorithm - These functions perform a depth first search
; trying to create a schedule that contains all required tasks.  First,
; the task durations are sorted to guarantee that they are in
; descending order. Next, the tasks are sorted by importance. This second
; step is important, since it insure low-importance tasks are degraded
; before high-importance tasks.
;
; The algorithm walks through each required task, trying to find a
; set of processor/task combinations that can be scheduled.  It starts
; by trying to schedule the maximum duration for the first task on the
; first processor of the system.  If it can't be scheduled on the first
; processor,  it tries the rest of the processors in turn.  If the first
; duration can't be schedule on any of the processors, it then
; repeats the process with the next task duration.  If no processor/
; task/durations can be scheduled, the algorithm returns nil.
;
; Once the first task is scheduled, the algorithm goes on to schedule
; each additional task in turn.  If at any point the system is unable to
; schedule a task, it will backup to the previous task, try the next
; task/processor/duration combination and then try to schedule the
; failed task again.  If the system is unable to schedule all tasks,
; the algorithm will return nil.
;---------------------------------------------------------------------------

(defun schedule-system(num-processors task-list)
   (let ((tlist (mapcar #'copy-task task-list)))
      (assign-priorities (schedule-system-aux tlist num-processors))))

(defun schedule-system-aux(tlist num-processors)
   (mapcar #'sort-task-duration tlist)
   (mapcar #'(lambda(x) (set-task-priority x 0)) tlist)
   (let ((system (build-system num-processors)))
      (schedule-tasks (sort-tasks-by-importance tlist) system)))

(defun schedule-tasks(task-list system)
   (cond ((null task-list) system)
         (t (let ((children
                    (generate-children (first task-list) (length system))))
                    (schedule-children (rest task-list) children system)))))

(defun schedule-children(task-list children system)
   (cond ((null children) nil)
         (t (let ((new-system (update-processor (first children) system)))
               (cond ((null new-system)
                        (schedule-children task-list (rest children) system))
                       (t (schedule-next-task task-list children system
                             new-system))))))

(defun schedule-next-task(task-list children old-system new-system)
   (let ((new-system (schedule-tasks task-list new-system)))
      (cond ((null new-system)
              (schedule-children task-list (rest children) old-system))
              (t new-system))))

;---------------------------------------------------------------------------
; Functions for generating all process combinations for a particular
; task.  These functions create a process for each task-duration-processor
; combination that can be supported by a particular task. Example:
; ((T1 10 0) (T1 10 1) (T1 10 2) (T1 5 0) (T1 5 1) (T1 5 2))
; The ordering of these children insures that first processors a
; checked, and then reduced durations are checked.
;---------------------------------------------------------------------------

(defun generate-children(task num-processors)
   (let ((children nil) (duration (task-duration task)))
      (generate-children-aux task duration num-processors
          nil num-processors)))
```

```
(defun generate-children-aux (task duration num-processors
                                   children processor)
   (cond ((null duration) children)
         ((not (equal processor '0))
          (generate-children-aux
              task
              duration
              num-processors
              (add-element-to-list
                 children (list task (first duration) processor))
              (- processor 1)))
          (t (generate-children-aux
                 task
                 (rest duration)
                 num-processors
                 children
                 num-processors))))

(defun update-processor(task system &optional (pos 1))
   (cond ((null system) nil)
         ((equal pos (get-processor task))
          (let ((new (add-element-to-list (first system) task)))
                (cond ((null (theorem1 new)) nil)
                      (t (cons new (rest system))))))
         (t (let ((new-system (update-processor task
                                   (rest system) (+ pos 1))))
             (cond ((null new-system) nil)
                   (t (cons (first system) new-system)))))))

;-------------------------------------------------------------------------
; Functions for Accessing Process Information
;-------------------------------------------------------------------------

(defun get-task(x)      (first x))
(defun get-duration(x)  (second x))
(defun get-processor(x) (third x))

;-------------------------------------------------------------------------
; Support Functions for the Scheduling Algorithm
;-------------------------------------------------------------------------

(defun add-element-to-list(l element)
   (append l (list element)))

(defun add-task-to-tasklist(l task)
   (add-element-to-list l task))

(defun sort-task-duration(task)
    (setf (task-duration task) (sort (task-duration task) #'>)))

(defun sort-tasks-by-importance(l)
        (sort l #'> :key #'(lambda(x) (task-importance x))))

(defun sort-processor-by-period(p)
        (sort p #'< :key #'(lambda(x) (task-period (get-task x)))))

(defun build-system(x)
   "Function for building system processor lists"
   (let ((system (list nil)))
      (dotimes (pos (- x 1) system) (setf system (cons nil system)))))

;-------------------------------------------------------------------------
; Functions for setting task priorities.  These functions are
; destructive in nature.  They change the actual task structure, they do
; not create a copy of the task.
;-------------------------------------------------------------------------
(defun set-task-priority(task priority)
   (setf (task-priority task) priority))

(defun assign-priorities(system)
   (dolist (processor system system)
     (when (not (null processor)) (assign-process-priorities processor))))
```

```
(defun assign-process-priorities(processor)
   (let ((priority MAX-PRIORITY))
       (dolist (process (sort-processor-by-period processor) t)
          (when (not (null process))
                (if (equal (get-duration process) 0.0)
                    (set-task-priority (get-task process) 0)
                    (set-task-priority (get-task process)
                        (incf priority)))))))

;-------------------------------------------------------------------------
; Rate-Monotonic Scheduling Thereom 1
;-------------------------------------------------------------------------
(defun utilization(task)
        (/ (get-duration task) (task-period (get-task task))))

(defun max-utilization(n)
   (* n (- (expt 2 (float (/ 1 n))) 1)))

(defun theorem1 (l &optional (u '0))
   (theorem1-aux l u (max-utilization (length l))))

(defun theorem1-aux(l u max)
   (cond ((null l) (if (< u max) u nil))
         (t (theorem1-aux (rest l)
                (float (+ u (utilization (first l)))) max))))
```

F.3.2 task.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; File:        task.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Description: This module is used by schedule.lisp to specify the
;;              task structure and provide a default set of tasks
;;              to be used in scheduling tests.
;;
;; See Also:    schedule.lisp, translate.lisp and print.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Data Structures:
;;   system     (processor processor processor ...)
;;   processor  (process process process ...)
;;   process    (task duration processor)
;;   task       (name duration period importance priority
;;               bodyin bodyout specin specout)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "USER")
(provide "task")

(defstruct task (name nil)
                (duration nil)
                (period nil)
                (importance 0)
                (priority   0)
                (bodyin  "./Template/template.adb")
                (bodyout "./Source/task_body.a")
                (specin  "./Template/template.ads")
                (specout "./Source/task_spec.a"))

(setf t1
    (make-task
       :name 'Status_Update
       :duration '(0.0030)
       :period '0.2000
       :importance '1
       :bodyout "./Source/task1.adb"
       :specout "./Source/task1.ads"))
```

```
(setf t2
    (make-task
       :name 'Keyset
       :duration '(0.0010)
       :period '0.2000
       :importance '3
       :bodyout "./Source/task2.adb"
       :specout "./Source/task2.ads"))

(setf t3
    (make-task
        :name 'Hook_Update
       :duration '(0.0020)
       :period '0.0650
       :importance '1
       :bodyout "./Source/task3.adb"
       :specout "./Source/task3.ads"))

(setf t4
    (make-task
       :name 'Graphic_Display
       :duration '(0.0090)
       :period '0.0800
       :importance '3
       :bodyout "./Source/task4.adb"
       :specout "./Source/task4.ads"))

(setf t4m
    (make-task
       :name 'Graphic_Display
       :duration '(0.009 0.008 0.007 0.006 0.005 0.004 0.003 0.002 0.001)
       :period '0.0800
       :importance '3
       :bodyout "./Source/task4.adb"
       :specout "./Source/task4.ads"))

(setf t5
    (make-task
        :name 'Stores_Update
       :duration '(0.0010)
       :period '0.2000
       :importance '1
       :bodyout "./Source/task5.adb"
       :specout "./Source/task5.ads"))

(setf t6
    (make-task
       :name 'Contact_Mgnt
       :duration '(0.0050)
       :period '0.0250
       :importance '2
       :bodyout "./Source/task6.adb"
       :specout "./Source/task6.ads"))

(setf t7
    (make-task
       :name 'Target_Update_I
       :duration '(0.0050)
       :period '0.0500
       :importance '1
       :bodyout "./Source/task7.adb"
       :specout "./Source/task7.ads"))

(setf t8
    (make-task
       :name 'Tracking_Filter
       :duration '(0.0020)
       :period '0.0250
       :importance '1
       :bodyout "./Source/task8.adb"
       :specout "./Source/task8.ads"))
```

```
(setf t9
    (make-task
       :name 'Nav_Update
       :duration '(0.0080)
       :period '0.0590
       :importance '1
       :bodyout "./Source/task9.adb"
       :specout "./Source/task9.ads"))

(setf t10
    (make-task
       :name 'Steering_Cmds
       :duration '(0.0030)
       :period '0.2000
       :importance '1
       :bodyout "./Source/task10.adb"
       :specout "./Source/task10.ads"))

(setf t11
    (make-task
       :name 'Nav_Status
       :duration '(0.0010)
       :period '1.0000
       :importance '1
       :bodyout "./Source/task11.adb"
       :specout "./Source/task11.ads"))

(setf t12
    (make-task
       :name 'Target_Update_II
       :duration '(0.0050)
       :period '0.1000
       :importance '3
       :bodyout "./Source/task12.adb"
       :specout "./Source/task12.ads"))

(setf t13
    (make-task
       :name 'Status_Update_II
       :duration '(0.0010)
       :period '1.0000
       :importance '1
       :bodyout "./Source/task13.adb"
       :specout "./Source/task13.ads"))

(setf t1-extra
    (make-task
       :name 'Collision_Monitor
       :duration '(0.0003)
       :period '0.200
       :importance '1
       :bodyout "./Source/task23.adb"
       :specout "./Source/task23.ads"))

(setf t2-extra
    (make-task
       :name 'Pilot_Manager
       :duration '(0.0081)
       :period '1.000
       :importance '1
       :bodyout "./Source/task24.adb"
       :specout "./Source/task24.ads"))

(setf t3-extra
    (make-task
       :name 'Tactical_Manager
       :duration '(0.0269)
       :period '1.000
       :importance '1
       :bodyout "./Source/task25.adb"
       :specin  "./Template/itask.ads"
       :specout "./Source/task25.ads"))
```

```
(setf t4-extra
    (make-task
       :name 'EOB_Manager
       :duration '(0.1331)
       :period '1.000
       :importance '1
       :bodyout "./Source/task26.adb"
       :specin  "./Template/itask.ads"
       :specout "./Source/task26.ads"))

(setf t5-extra
    (make-task
       :name 'File_Manager
       :duration '(0.0026)
       :period '1.000
       :importance '1
       :bodyout "./Source/task27.adb"
       :specout "./Source/task27.ads"))

(setf t6-extra
    (make-task
       :name 'Profile_Manager
       :duration '(0.0013)
       :period '1.000
       :importance '1
       :bodyout "./Source/task28.adb"
       :specout "./Source/task28.ads"))

(setf t7-extra
    (make-task
       :name 'IFF_Processor
       :duration '(0.0010)
       :period '1.000
       :importance '1
       :bodyout "./Source/task29.adb"
       :specout "./Source/task29.ads"))

(setf t8-extra
    (make-task
       :name 'Waypoint_Sequencer
       :duration '(0.0006)
       :period '1.000
       :importance '1
       :bodyout "./Source/task30.adb"
       :specout "./Source/task30.ads"))

(setf t9-extra
    (make-task
       :name 'Track_Handler
       :duration '(0.005)
       :period '0.100
       :importance '1
       :bodyout "./Source/task19.adb"
       :specout "./Source/task19.ads"))

(setf t10-extra
    (make-task
       :name 'Track_Manager
       :duration '(0.0012)
       :period '0.100
       :importance '1
       :bodyout "./Source/task21.adb"
       :specout "./Source/task21.ads"))

(setf t11-extra
    (make-task
       :name 'Response_Manager
       :duration '(0.0009)
       :period '0.100
       :importance '1
       :bodyout "./Source/task22.adb"
       :specout "./Source/task22.ads"))
```

Bibliography

[Agrawala, 1994]. Agrawala, Ashok. Computer Science Department, University of Maryland, College Park, Personal interview, 7 September 1994.

[Baker, 1989]. Baker, T.P., and Alan Shaw. "The Cyclic Executive Model and Ada," *The Journal of Real-Time Systems*, 1:7-25, June 1989.

[Dodhiawala, 1988]. Dodhiawala, Rajendra and N. S. Sridharan. *Real-Time Impact Report (RT-1 Impact Analysis)*. MCAIR SDRL 10-1. Santa Clara, CA: FMC Corporation, Central Engineering Laboratories, September 1988.

[Gschwendtner, 1992]. Gschwendtner, A. B. *DARPA Neural Network Study*. AFCEA Press, Fairfax, Virginia., 1992.

[Hoogeboom, 1992]. Hoogeboom, Boudewijn and Wolfgang A. Halang. "The Concept of Time in the Specification of Real-Time Systems." *Real-Time Systems - Abstrations, Languages, and Design Methodologies,* edited by Krishna M. Kavi, 19-38, IEEE Computer Society, 1992.

[Locke, 1990]. Locke, C. Douglass, David R. Vogel, and Thomas J. Mesler. "Predictable Real-Time Avionics Design Using Ada Tasks and Rendezvous: A Case Study." *Ada Letters*, X(9):118-120, Fall 1990.

[Locke, 1992]. Locke, C. Douglass. "Software Architecture for Hard Real-Time Applications: Cyclic Executives vs Fixed Priority Executives," *The Journal of Real-Time Systems*, 4:37-53, April 1992.

[Liu, 1973]. Liu, C. L., and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *The Journal of the Association for Computing Machinery*, 20(1):46-61, January 1973.

[Musliner, 1993]. Musliner, David J. and others. "CIRCA: A Cooperative Intelligent Real-Time Control Architecture," *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6):1561-1573, November/December 1993.

[Musliner, 1994]. Musliner, David J. and others. *The Challenges of Real-Time AI*. Technical Report, University of Maryland, CS-TR-3290, June 1994.

[Raeth, 1994]. Raeth, P. G. WL/FIPA, Wright Laboratories, Flight Dynamics Directorate, Wright-Patterson AFB, OH 45433, Personal interview, 24 Jan 1994.

[Shamsudin, 1991]. Shamsudin, Annie Z. and T. S. Dillion. *T.S. NetManager: A Real-Time Expert System for Network Traffic Management*. Technical Report 15/91. Department of Computer Engineering, La Trobe University, Bundoora, Victoria, Australia, December 1991.

[Stankovic, 1988]. Stankovic, John A. "Misconceptions About Real-Time Computing," *IEEE Computer*, 10-19, October 1988.

[Stankovic, 1990]. Stankovic, John A. *The Spring Architecture*. Technical Report, University of Massachusetts, UM-CS-1990-026, June 1990.

[Stankovic, 1993a]. Stankovic, John A. *On the Reflective Nature of the Spring Kernel.* Technical Report, University of Massachusetts, UM-CS-1990-119, 1993.

[Stankovic, 1993b]. Stankovic, John A. *Reflective Real-Time Systems.* Technical Report, University of Massachusetts, June 1993.

[Whelan, 1992]. Whelan, Michael Anthony. *An Intelligent Real-Time System Architecture Implemented in Ada.* MS Thesis, AFIT/GCE/ENG/92D-12, Air Force Institute of Technology, School of Engineering, Wright-Patterson AFB, OH, December 1992.

[Xu, 1991]. Xu, Jia and David Lorge Parnas. "On Satisfying Timing Constraints in Hard-Real-Time Systems", 132-146, November 1991.

Vita

Captain Robert E. J. Caley was born in London, England on 2 May 1964. At the age of twelve he moved to the United States where he attended several high schools in New York, New Jersey, and Massachusetts. After graduating from Walpole High School in 1981, at the age of seventeen, he became an enlisted member of the United States Air Force. Three years as airman convinced Captain Caley that he wanted to become an officer.

Captain Caley earned his United States citizenship in 1984. Following this, he was accepted as a student at the Air Force Academy Preparatory school. One year later, he was a cadet at one of the most prestigious universities in the country, the United States Air Force Academy.

In 1989, Captain Caley realized his dream of becoming an officer, when he graduated from the Air Force Academy with a degree in computer science and a commission in the United States Air Force. Upon graduation, Captain Caley was assigned to the Air Force Military Personnel Center (AFMPC), Randolph AFB, TX, where he served as a software program analyst for Personnel Concept III (PC-III).

Captain Caley transferred to Wright-Patterson AFB, OH in May 1993, to attend the Air Force Institute of Technology (AFIT) and earn a Master of Science (Computer Science). Following graduation, Captain Caley will be assigned to the Air Intelligence Agency (AIA), where he will once again perform duties as a software program analyst.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
Next-Generation Real-Time Systems:
Investigating the Potential of Partial-Solution Tasks

5. FUNDING NUMBERS

6. AUTHOR(S)
Robert E. J. Caley

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/94D-01

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Major Peter G. Raeth
WL/FIPA
Wright Laboratory, Flight Dynamics Directorate
Wright-Patterson AFB, OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

While the cyclic executive and fixed-priority scheduling strategies have been sufficient to handle traditional real-time requirements, they are insufficient for dealing with the complexities of next-generation real-time systems. New methods of intelligent control must be developed for guaranteeing on-time task completion for real-time systems that are faced with unpredictable and dynamically changing requirements. Implementing real-time processes as partial-solution tasks is one technique that may be beneficial. This type of task, when combined with intelligent control, has the potential for increasing pre-runtime schedulability, system maintainability, and runtime robustness. This research investigates the benefits of partial-solution tasks by experimentally measuring the change in performance of 11 simulated real-time systems when converted from all-or-nothing tasks to partial-solution tasks. Results from the experiments indicate that partial-solution tasks have the potential to decrease missed deadlines and increase a systems' average solution quality. The results also suggest that best performance gains can be achieved using Optimistic partial-solution tasks where the bulk of solution quality is achieved early during task execution. The framework used in this research was developed to measure the general case performance characteristics of partial-solution tasks. As a by-product, the research resulted in a framework that can also be used to measure specific case characteristics.

14. SUBJECT TERMS
Real-Time Systems, Intelligent Control, Next-Generation Systems

15. NUMBER OF PAGES
135

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL